THESIS

PROBABILITY DISTRIBUTIONS OVER CRYPTOGRAPHIC PROTOCOLS

by

Stephanie J.C. Skaff

June 2009

Thesis Advisor:	Jonathan Herzog
Second Reader:	George Dinolt

1. REPORT DATE *(DD–MM–YYYY)* 19-06-2009	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* 24-09-2007 – 19-06-2009

4. TITLE AND SUBTITLE Probability Distributions over Cryptographic Protocols	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER DUE-0414102
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Stephanie J.C. Skaff	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Department of the Navy	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This study examines the problem of assuring correct functionality of cryptographic protocol verifiers. As a replacement for manual input of well-known protocols, we propose the creation of a random protocol generator capable of producing protocols of varying degrees of correctness. This generator would be verifier-independent, and the protocols translated into verifier languages as required. This would automate not only the creation of protocols, but eliminate the variability both in translation quality and in the resulting body of tests. To this end, we propose a common definition for cryptographic protocols, develop multiple probability distributions over this definition, and implement a generator that uses these distributions. As a proof of concept, we translate protocols created by the generator into a suitable format for the Cryptographic Protocol Shapes Analyzer.

15. SUBJECT TERMS

automatic protocol generation; protocol analysis; security protocols; cryptographic protocols; key-exchange protocols; authentication protocols; protocol verification

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(include area code)*
Unclassified	Unclassified	Unclassified	UU	153	

THIS PAGE INTENTIONALLY LEFT BLANK

PROBABILITY DISTRIBUTIONS OVER CRYPTOGRAPHIC PROTOCOLS

Stephanie J.C. Skaff
Civilian, Naval Postgraduate School
B.A., College of St. Benedict, 2003

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2009

Author: Stephanie J.C. Skaff

Approved by: Jonathan Herzog
Thesis Advisor

George Dinolt
Second Reader

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This study examines the problem of assuring correct functionality of cryptographic protocol verifiers. As a replacement for manual input of well-known protocols, we propose the creation of a random protocol generator capable of producing protocols of varying degrees of correctness. This generator would be verifier-independent, and the protocols translated into verifier languages as required. This would automate not only the creation of protocols, but eliminate the variability both in translation quality and in the resulting body of tests. To this end, we propose a common definition for cryptographic protocols, develop multiple probability distributions over this definition, and implement a generator that uses these distributions. As a proof of concept, we translate protocols created by the generator into a suitable format for the Cryptographic Protocol Shapes Analyzer.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgements

I would like to thank the Naval Postgraduate School Department of Computer Science, the Center for Information Systems Security Studies and Research, and the National Science Foundation for their support and sponsorship. This material is based on work supported by the National Science Foundation under Grant DUE-0414102. Special thanks go to my second reader, Dr. George Dinolt, for his encouragement and good humor throughout my time at NPS.

I owe a profound debt of thanks to my advisor, Dr. Jonathan Herzog, whose patience, wit, and enthusiasm about the subject matter led me to this thesis and through to its completion. Thank you for the most frustrating and fascinating experience of my life thus far.

Finally, I would like to thank my spouse, John "Garick" Chamberlin, whose love and support were invaluable during my graduate work and the resulting long separation. I hope this has been worth it.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1

Introduction

1.1 Background

In any public network, participants communicate across an insecure medium–and, as in any public venue, not all the inhabitants are well-intentioned. Any malicious entity with sufficient resources may intercept, eavesdrop, and counterfeit messages on the network to its own ends. Intruders may compromise the confidentiality of the communication (*e.g.,* eavesdropping on messages to learn secret or private information). They may also attack the integrity of the communication by impersonating one party to another (*e.g.,* misrepresenting themselves to a bank as a legitimate customer).

Successful attacks can be costly. In some cases, such as damage to reputation, the cost may be difficult to quantify. In others, the cost may be directly reckoned in monetary damage or in human lives. Even a small-scale theft, when perpetrated on the scale of large international networks, may yield a significant gain. This economy of scale has made attempts to discover and exploit flaws in communication protocols a worthwhile endeavor with only limited risk to an attacker.

To prevent such attacks, sensitive communication usually begins with a *cryptographic protocol* (also referred to as a *security protocol*). A cryptographic protocol is a sequence of messages that provides assurance to each participant about the other participants. Common security goals for protocols include assurance of authentication (a participant is the party she claims to be) and secrecy (protected information cannot be read by parties not participating in the protocol). Authentication protects the participants from impersonation, while secrecy limits the damage done by eavesdropping.

While they may seem easy to create, security protocols that function correctly despite the actions of a malicious party are surprisingly more difficult to design successfully. Researchers have discovered several flaws in the design and implementation of cryptographic algorithms, which have led to successful message decryption (*e.g.,* Microsoft's implementation of RC4 [1], the Debian Linux implementation of OpenSSL [2]).

In this thesis, however, we address the problem of *structural* flaws, in which the protocol messages themselves may be used to compromise security without breaking the underlying cryptography. Perhaps the most well-known example of a protocol with structural flaws is the Needham-Schroeder Public-Key Authentication Protocol [3], diagrammed in Figure 1.1.

1. $A \rightarrow B : \{A, N_a\}_{K_b}$
2. $B \rightarrow A : \{N_a, N_b\}_{K_a}$
3. $A \rightarrow B : \{N_b\}_{K_b}$

Figure 1.1: The Needham-Schroeder public key protocol

Participant *A* initiates the protocol by sending to participant *B* a message containing *A*'s name and a freshly-generated value (*nonce*), both encrypted with *B*'s public key. *B* replies with *A*'s nonce and a nonce of *B*'s own, both encrypted with *A*'s public key. *A* completes the exchange by sending *B*'s nonce back to *B*, encrypted with *B*'s public key. At the end of this exchange, each participant has read a message that could only be decrypted by someone who hold that participant's private key; thus, the reasoning goes, the participants are really who they claim to be.

This protocol, published in 1978, was in use for seventeen years before Lowe discovered a weakness that allowed a simple man-in-the-middle attack [4]. In this attack, shown in Figure 1.2, an honest participant *A* initiates an instance of the protocol with a dishonest participant *E*. Unbeknownst to *A*, *E* uses the information in the initiation message to begin a second and simultaneous instance of the protocol in which it masquerades as *A* to a third party *B*. *B* responds to *E* with the expected response message encrypted with *A*'s public key. *E* cannot read this message, but forwards it to *A*. Since this message does not include the identity of the responder, *A* cannot tell that the response was originally created by *B* rather than by *E*. Thus, *A* does not know that *B* is attempting to participate with *A* in a protocol run. Once both protocol instances are complete, *A* is in communication with *E*, as intended. *B*, however, believes it is in communication with *A* when it is actually in communication with *E*. Further, both *A* and *B* erroneously believe that the nonces N_a and N_b are known to the honest participants alone.

1. $A \rightarrow \quad E : \{A, N_a\}_{K_e}$
2. $E(A) \rightarrow \quad B : \{A, N_a\}_{K_b}$
3. $B \rightarrow E(A) : \{N_a, N_b\}_{K_a}$
4. $E \rightarrow \quad A : \{N_a, N_b\}_{K_a}$
5. $A \rightarrow \quad E : \{N_b\}_{K_e}$
6. $E(A) \rightarrow \quad B : \{N_b\}_{K_b}$

Figure 1.2: Lowe's attack on the Needham-Schroeder public key protocol

This flaw is far from alone. As another example, Mao and Boyd [5] uncovered a flaw in the mutual authentication protocol proposed by Otway and Rees [6]. This flaw allows an attacker to successfully impersonate one party to another by intercepting and replaying a message fragment from a previous exchange between them. More recently, a flaw in the implementation of the OpenSSL SSL/TLS server allowed the use of version rollback attacks, in which SSL participants are forced to agree on their weakest common cryptographic suite rather than their strongest [7].

The difficulties notwithstanding, new applications and technology constantly require creation and extension of protocols. During the week of 17-23 February 2008 alone, the Internet Engineering Task Force received 230 new Internet Drafts for consideration. Of these, 60 contained either proposals for new security protocols or requests to extend or modify existing protocols. New protocols are proposed when no suitable protocol already exists. Modifications of existing protocols are usually either adaptations of protocols for new purposes or responses to discovery of vulnerabilities in protocols already in use.

With this level of demand, the protocol design process would clearly benefit from some level of automation. Two types of tools exist to assist protocol designers: protocol *generators* and protocol *verifiers*.

Automated security protocol generation is an active area of research [8, 9, 10, 11]. This approach appears promising, as it captures the expertise of experts for use by application designers. However, protocol generators have thus far been of limited use beyond the cryptographic protocol research field. Application designers do not generally follow cryptographic protocol research, and are unlikely to be aware of either the existence of such generators or of the pitfalls of protocol design should they attempt to create a protocol of their own.

Computer security specialists, on the other hand, focus most of their attention beyond the creation of a protocol onto its correctness in meeting security needs. For this purpose, they use protocol *verifiers*, which inspect protocols for possible flaws.

1.2 Problem Statement

The use of a protocol verifier shifts the assurance burden of protocol analysis to the verification tool. Given the potential costs of a compromise, we would like some level of assurance that the verifier will correctly identify all structural flaws of a protocol. This requires that the verifiers themselves be evaluated and verified for correctness. The current level of rigor for such evaluation is unfortunately uneven. In most cases, the common evaluation practice is to translate a small number of well-studied protocols into the verifier input language, then confirm that the verifier discovers at least the known attacks against each protocol. This body of test cases ranges from under ten protocols at worst to approximately fifty at best, depending on the capabilities of the verifier and the effort expended by the verifier designers.

The scarcity of test cases can be attributed to the manual method used to produce them. First, a sufficient body of test protocols must be either created or adapted from existing proposals. These protocols must then be expressed as input for the verification tools under study, most of which have their own input language. This creation and translation process is both tedious and prone to human interpretation error. Moreover, most protocol researchers specialize in only a small subset of verifiers; thus, translation of protocols for several verifiers by a single person to maintain continuity of assumptions may produce varying quality of translations, rendering comparisons suspect. Given the number of cases needed for robust verifier testing, manual creation and translation of protocols quickly becomes infeasible.

Automated protocol generators are certainly useful in generation of test cases for verification tools. Existing generators such as the Automated Protocol Generator [9] and the Automatic Synthesis Protocol Builder [10], however, proceed from the assumption that the desired end result is a secure protocol, and limit their generation accordingly. While production of secure protocols is the desired end result, assuring the correctness of verification tools for those protocols requires a body of test cases with greater variety. Additionally, most generators produce output for a specific verifier, which precludes using a single set of test cases against multiple verifiers.

In this work, therefore, we consider instead the creation of a *random* generator capable of producing protocols of varying degrees of correctness. Ideally, such a generator would be tool-independent; additional modules could then provide translation services into verifier languages as required. This would automate not only the creation of protocols, but eliminate the variability both in translation quality and in the resulting body of tests.

Creation of a random protocol generator requires that we first answer three questions: first, what is a protocol? Second, what is a *random* protocol? Third, how random should a protocol be? Answering these questions yields a working definition of a cryptographic protocol, the degrees of freedom in the definition, and at least one probability distribution over the protocol degrees of freedom.

1.3 Contributions

In this work, we develop multiple probability distributions over the set of cryptographic protocols. In particular, we propose a common definition for cryptographic protocols, develop probability distributions over this definition (three naive and one realistic), and implement a protocol generator that uses these distributions.

In more detail, we first examine existing definitions of cryptographic protocols, noting their common ground and their differences. From these, we formulate a common definition of a protocol. We treat areas of agreement as given and reconcile disagreements as necessary. Some disagreements result from the process of protocol analysis rather than generation; in these cases, no choice is necessary. In the case of a disagreement related to generation, we choose the most general option.

We next create probability distributions over the common protocol definition. To do this, we first identify the degrees of freedom in the protocol definition and the set of valid values for each degree. Given these, we propose both naive and realistic probability distributions over this set. The naive distribution utilizes all degrees of freedom, sampling uniformly from finite sets of choices and exponentionally from infinite sets of choices. To generate the realistic distribution, we analyze a sample of protocols from the Clark-Jacob protocol library [12] for both general and statistical properties of real-world protocols. We then use the results of the analysis to limit the degrees of freedom in two ways. First, we remove degrees of freedom by introducing new assumptions on the protocol definition based on the general protocol properties found. Second, we constrain the remaining degrees of freedom using a Gaussian distribution based upon on the statistical properties of the sample.

We design and implement a generator that uses these distributions to create protocols. Implementation of both the naive and realistic distributions follows naturally from the distribution design. Run-time behavior of the realistic distribution is similarly straightforward. Due to run-time issues with the naive distribution, we introduce two limited versions of the naive distribution. These new distributions significantly improve the run-time behavior in the naive case. Finally, we validate our generator by evaluating the generated protocols (suitably translated) using the Cryptographic Protocol Shapes Analyzer (CPSA) [13].

The rest of this work is as follows. Chapter 2 of this thesis reviews previous work in this area and the basis for the current study. Chapter 3 proposes design criteria for the random generation of cryptographic protocols, while Chapter 4 discusses probability distributions over that definition. In Chapter 5, we discuss the implementation of the protocol generator and a sample translation module. In Chapter 6, we summarize our results and provide suggestions for future work in this area.

CHAPTER 2

Previous Work

At this time, we are unaware of an automated protocol generator that is not biased toward correct protocols. To the best of our knowledge, this work represents the first attempt to deliberately generate protocols for the specific purpose of verification tool testing.

In this chapter, we compare earlier research in automatic protocol generation with the current work.

2.1 Evolutionary Search

Clark and Jacob [14, 8] implemented an automated protocol generator whose output could be analyzed using the Burrows-Abadi-Needham (BAN) logic for protocol analysis [15]. BAN logic consists of a set of rules by which analysts may reason about identities of participants, the beliefs those participants hold, characteristics of messages sent and received, and so forth. Protocols are defined in the BAN logic as messages from one participant to another, and analysis depends on ascertaining the beliefs each participant holds. To analyze a protocol using the BAN logic, one first converts the actual messages of the protocol into an idealized form that supposedly captures all essential attributes of each message. The analyst then annotates the protocol to describe the beliefs of each participant as regards the other participants before and after each message.

Finally, the analyst may use the BAN inference rules to analyze the protocol. All participants in an exchange are assumed to be honest; all messages are considered subject to interception and alteration.

The Clark-Jacob generator used simulated annealing [16] and genetic evolutionary algorithms [17] to form protocols. The original work was limited to the symmetric key portions of the BAN logic; later research extended this to include asymmetric encryption and hybrid encryption schemes, allowing the use of the full BAN logic [18]. As with the BAN logic itself, this generator creates only idealized protocols, leaving concrete refinement to be performed as a manual process. This is somewhat unsatisfactory, as one of the main benefits of automating protocol design is to remove individual interpretation and error from the process. Further, the Clark-Jacob generator is designed to produce hypothetically correct protocols. It is therefore unlikely to generate protocols that contain unusual cases or significant coverage of flaws. Our work, on the other hand, produces a more comprehensive body of test protocols.

2.2 Athena Automated Protocol Generator

Concurrent with but independently of Clark and Jacob, Perrig and Song [19] proposed the Athena protocol generator (APG) and verifier (APV) based on extensions to the strand space model [20]. In this model, a protocol is described in terms of *strands* - a sequence of messages sent and received by a participant, with no indication of the participants on the other end of the transmission. A *strand space* consists of the strands of each legitimate participant in the protocol and additional strands representing the actions of adversaries. Should a message be sent by one participant strand and received by another, the exchange links those strandss into a *bundle*.

Song's automatic protocol generator (APG) [9] creates candidate protocols in accordance with user-selected parameters (key type, number of participants, desired goals, *et cetera*). These protocols may then be analyzed by APV [21]. APG uses iterative deepening to search for candidate protocols within a user-specified cost metric. It also performs state-space pruning to remove search branches that clearly do not lead to usable protocol bundles or consist of permutations of already-generated states. Like that of Clark-Jacob, this approach produces protocols that tend toward correctness, and is less suitable for producing a broad set of test cases. In contrast, our generator produces protocols that widely varying degrees of correctness, and is therefore of more utility for verifier testing.

2.3 Automatic Synthesis Protocol Builder

Zhou and Foley [10] found both previous approaches to protocol generation to be insufficient for their needs. The BAN-based evolutionary search techniques relied on specification of a useful fitness function; Clark and Jacob themselves noted that creating a highly accurate fitness function would be problematic at best. The Athena system, on the other hand, is completely based in the strand space model, and would therefore require major modifications to support Zhou and Foley's BAN-based methodology. They instead proposed the Automatic Synthesis Protocol Builder (ASPB) as an attempt to combine the two approaches. APSB uses the Buttyán-Staamann-Wilhelm extensions [22] to the BAN logic, which maintain the simplicity of BAN logic while removing assumptions about the trustworthiness of participants.

Zhou and Foley combined this approach with Guttman's authentication test design method [23] to direct a backwards search from a stated end goal to protocols meeting that end goal. Unlike our work, the sole goal of ASPB is to produce provably secure protocols. As such, it is also ill-suited to the production of a large and varied body of test protocols.

2.4 Artificial Immune Algorithm

Xue *et al.* [11] developed a protocol generator based on the application of of cord calculus [24] (a variant on π-Calculus [25]) as applied to artificial immune algorithms. This approach uses security goals and running environment as initial antigens in an immune system simulation. Messages are treated as antibodies, which undergo crossover and mutation until they satisfy an affinity function against the security goals.

As with the generators above, generation using the artificial immune algorithm is intended to produce secure protocols, and is thus less than ideal for generation of test protocols.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3

Design Decisions

In this chapter, we propose design criteria for the random generation of cryptographic protocols. To accomplish this, we first describe the assumptions common to most definitions of security protocols, extracting from those assumptions a common definition of a security protocol. We then evaluate the original definitions' differences to determine their potential effect on the protocol generation process and refine our design criteria accordingly.

3.1 Common Ground

We begin by considering previous protocol definitions to determine their commonalities and differences. Existing definitions may be found in protocol specification languages (*e.g.,* the Common Authentication Protocol Specification Language (CAPSL) [26], the High Level Protocol Specification Language (HLPSL) [27]) and in the input languages for protocol verifiers.

Most definitions of a security protocol agree on several points, which fall into two categories: assumptions about the operating environment and assumptions about the protocol design itself. Agreed-upon assumptions about the operating environment are as follows:

- All participants in a protocol assume that they communicate across an adversarial network. Any transmitted message will be overheard and may be intercepted, altered, or replayed.

- Protocol participants may share pre-existing secrets not known by the adversary.

- One or more of the participants may be a certificate server or other trusted third party.

- Participants are able to reliably generate random values that are both unlikely to have previously been used and unlikely to be guessed by an adversary.

- Multiple protocol runs by the same entity, whether simultaneous or not, are independent. While they share the same initial knowledge (*e.g.,* long-term keys), each acts as though no other protocol runs exist.

Similarly, protocol analysis tools generally agree on these design assumptions:

- A protocol consists of a set of two or more roles that may interact via message transmission and reception.

- Messages contain one or more component atoms. These include names of participants, nonces (random values not heard before the current run of a protocol), and some representation of cryptographic keys.

- Messages may also contain one or more operations to be taken on components. The most common operations are concatenation and encryption; designers may also define other operations such as hashing and message signing. Operations may be either explicit or implied by the structure of a message.

- Messages may be represented either symbolically or by using a parse tree.

- Protocols are constructed using a free algebra. A given message may only be built or parsed in one way.

- The actions taken by a roles in a protocol follow a straight line of execution, and do not branch based on input received.

We demonstrate these protocol assumptions with the Otway-Rees protocol [6], as found in the Clark-Jacob protocol library [12]. In this example, A and B are two correspondents, and S is a trusted key server.

1. $A \rightarrow B : M, A, B, \{N_a, M, A, B\}_{K_{as}}$
2. $B \rightarrow S : M, A, B, \{N_a, M, A, B\}_{K_{as}}, \{N_b, M, A, B\}_{K_{as}}$
3. $S \rightarrow B : M, \{N_a, K_{ab}\}_{K_{as}}, \{N_b, K_{ab}\}_{K_{bs}}$
4. $B \rightarrow A : M, \{N_a, K_{ab}\}_{K_{as}}$

Figure 3.1: The Otway-Rees Protocol

As noted above, this protocol has three participants connected by a sequence of sent and received messages. Note that A never directly communicates with S; this demonstrates that a participant in a protocol need not communicate directly with all other participants. The component atoms of this protocol are symbolically represented by the names A, B, and S; the nonces N_a and N_b; and the keys K_{as}, K_{bs}, and K_{ab}. (M is a plaintext chosen by A and transmitted in the clear. Since it can be easily overheard and reproduced, this type of plaintext component is generally ignored in structural analysis of proptocols. We include it here for the sake of accurately reproducing the protocol.) Message encryption and signature are specified by enclosing the encrypted/signed items in brackets followed by the encryption or signature key (*i.e.,* $\{x\}_{K_z}$, where x is encrypted with the key K_z.) Signature keys may be implicitly distinguished from encryption keys by the notation "$^{-1}$", which denotes a private key. Concatenation is implicitly represented in the specification by a comma between atoms.

With the exception of the first message, each message may only be constructed once the previous message has been received. For example, B cannot send message four to A until it has received the message from S that contains $\{N_a, K_{ab}\}_{K_{as}}$. Further, at no point do any of the participants have a choice of messages to send in order to continue the protocol run.

3.1.1 Composite Definition

Based on these assumptions and the examined definitions themselves, we define a cryptographic protocol in Figure 3.2.

- *Protocol* : set of roles
- *Role* : sequence of nodes containing messages sent and received when executing a role
- *Node* : encapsulates a message; contains direction of message from the perspective of the role(sent or received) and message contents
- *Message* : sequence of atoms, encryptions of messages, and concatentations of messages
- *Encryption* : a message encrypted by a key
- *Concatenation* : joins a message to another message
- *Atom* : values in a message. These may be participant names, nonces, and keys
- *Keys* : symmetric (long-term shared keys and session keys) or asymmetric (public/private key pairs)
- *Encrypting keys* : public keys and any symmetric keys

Figure 3.2: Common definition of a cryptographic protocol

This list contains the elements common to all protocol definitions examined. As mentioned above, some protocol definitions define additional message operations such as hash functions or digital signatures; these are not captured in the common definition. Though such operations may be sufficiently widespread in future protocol definitions to warrant inclusion on a composite definition of a protocol, we do not include them at this time.

3.2 Disagreements

While protocol verifiers find common ground in environmental assumptions and basic composition, they differ in several aspects of the protocol analysis process. We examine each of these issues in terms of its impact on the protocol generation process. In this section, we address five areas of disagreement: analysis procedures used, independence of roles, deconstruction of messages, specification of goals, and the scope of the analysis search space.

3.2.1 Analysis Procedure

Dolev and Yao [28] first proposed using a formal model (also commonly called the Dolev-Yao model) for cryptographic protocol analysis. Analysis in the Dolev-Yao model is based on an external view of the protocol run; the analyst proves theorems about what additional messages may be sent or received given the messages transmitted in the protocol.

Belief logics [15, 29, 22], on the other hand, view the protocol from the perspective of each of its participants. Protocol analysis using belief logic involves application of a system of inference rules upon messages between participants. The results of this application are used to reason about each participant's beliefs about other participants' knowledge and intentions. Should all honest participants hold the same set of beliefs about the other participants at the completion of the protocol run, the protocol is considered secure. In the following example of inference rule use, the symbol $\mid\equiv$ represents the inference rule "believes", $\mid\Rightarrow$ represents the speaking of a message, and $X \overset{Z}{\longleftrightarrow} Y$ represents communication between two participants *X* and *Y* using key *Z*.

$A \mid\equiv S \mid\Rightarrow A \overset{K}{\longleftrightarrow} B$

This may be interpreted as "*A* believes *S* has said that *A* and *B* may communicate using *K*."

While most protocol verifiers base their analysis on the Dolev-Yao model [30, 21, 31], at least one verifier [32] performs automated BAN analysis. Since the beliefs associated with a protocol are a product of its analysis, they are out of scope for the protocol generation process. Our generator is capable of producing protocols useful in either system given suitable translation to a specific verifier input language.

3.2.2 Independence of Participants

In both the original Dolev-Yao model and in belief logic, a protocol is described as a sequence of messages between protocol participants. The sequence of messages describes an *intended run* of the protocol. As an example, we examine a simplified version of the Needham-Schroeder public key protocol [3].

1. $A \rightarrow B : \{A, N_a\}_{K_b}$
2. $B \rightarrow A : \{N_a, N_b\}_{K_a}$
3. $A \rightarrow B : \{N_b\}_{K_b}$

Figure 3.3: Intended-run view of the simplified Needham-Schroeder public key protocol

In the first message, *A* sends to *B* a message composed of *A*'s name and a nonce generated by *A*, all encrypted using *B*'s public key. In the second message, *B* returns *A*'s nonce with a new nonce generated by *B*, encrypted using *A*'s public key. The final message in the protocol is from *A* to *B*, consisting of *B*'s nonce encrypted using *B*'s public key.

The Common Authentication Protocol Specification Language (CAPSL) [26] and the input language for Casper [31] are examples of languages that describe protocols in terms of an intended run.

Alternately, protocols may be formulated in terms of *independent processes*. In this method, a message is described both by its contents and by the direction it travels from a given process (*i.e.,* sent or received). In this view, the Needham-Schroeder protocol appears as follows:

$$\begin{tikzcd}
A & B \\
-\{A, N_a\}_{K_b} \arrow[r] \arrow[d, Rightarrow] & +\{A, N_a\}_{K_b} \arrow[d, Rightarrow] \\
+\{N_a, N_b\}_{K_a} \arrow[d, Rightarrow] & -\{N_a, N_b\}_{K_a} \arrow[l] \arrow[d, Rightarrow] \\
-\ \{N_b\}_{K_b} \arrow[r] & +\{N_b\}_{K_b}
\end{tikzcd}$$

Figure 3.4: Inpendendent process view of the simplified Needham-Schroeder public key protocol

As can be seen in the diagram, each role sends and receives the same messages as in the intended-run view. The participant in a given role, however, makes no assumptions about the identity of a participant on the other end of a message, but relies only on the contents of the message to provide clues about the identities of correspondents.

This separation is made explicit in protocol specifications for verifiers based on the independent-processes view. These specifications are organized in terms of each role and its list of messages sent and received, as seen in the CPSA specification of the Needham-Schroeder in Figure 3.5.

```
(defprotocol ns basic
   (defrole init
      (vars (a name) (b name) (n-a text) (n-b text))
      (trace
         (send(enc (cat n-a a) (pubk b)))
         (recv(enc (cat n-a n-b) (pubk a)))
         (send (enc n-b (pubk b)))))
   (defrole resp
      (vars (b name) (a name) (n-b text) (n-a text))
         (trace
            (recv (enc (cat n-a a) (pubk b)))
            (send (enc (cat n-a n-b) (pubk a)))
            (recv (enc n-b (pubk b))))))
```

Figure 3.5: CPSA specification of the Needham-Schroeder protocol

In this specification, the *defprotocol* line specifies the protocol name (*ns*) and the algebra used in the specification (*basic*). For each role, the specification includes a list of variables and a trace of the messages sent and received by that role. Unlike the previous protocol examples, the trace indicates only the trace and content of each message, and does not indicate the recipient.

Analysis techniques based on the latter approach include Spi calculus [33] (a variant on π-Calculus [25]) and strand space [20] (described in section 2.2).

The High Level Protocol Specification Language (HLPSL) created by the Automated Validation of Internet Security Protocols and Applications (AVISPA) project [27] combines both approaches. An HLPSL specification contains an specification for each role and a composite role that combines the individual roles into an intended run.

It is possible to randomly generate a protocol with an intended run and re-express it in terms of independent processes without loss of randomness in terms of participants. The converse, however, is not necessarily true. We therefore implement the independent approach in the generator as part of the most random probability density function, and implement the intended run in the more realistic probability density function.

3.2.3 Implicit vs. Explicit Deconstruction

Most verifiers use pattern matching to implicitly model decomposition and decryption of messages. For example, the message

– $\{A, X\}_{K_b}$

represents two items encrypted with B's public key. The message is implicitly decrypted by any participant who possesses the private key K_b^{-1}.

Other verifiers take a different approach. The input language for ProVerif [30], for example, explicitly specifies decryption and deconstruction operations as follows:

in (c,m);
let (A, NA) = decrypt(M, skA) in ...

In this example, the protocol particant receives a message in the first line and decrypts it to retrieve its contents in the second.

In general, explicit deconstruction and decryption may be found in verifiers based on the spi calculus [34], such as ProVerif. The Cryptographic Protocol Type Checker (Cryptyc) [35], however, is a notable exception. Cryptyc integrates use of pattern-matching in the spi calculus framework, which in turn allows the specification of nested cryptographic primitives.

Our generator takes the first approach, implicitly modelling decryption and deconstruction operations. Since explicit deconstruction and decryption operations are not required by all verifiers, insertion of necessary explicit operations for those that do is left to the process of translation into the verifier's input language.

3.2.4 Treatment of Goals

Verifiers may or may not require inclusion of security goals in protocol specifications. Some verifiers, such as Athena [21], will not analysize a protocol without the inclusion of goals. Others, such as CPSA [13], accept protocols with no specified goal as input. Analysis in the latter case may not produce useful information on protocol security, but is useful in determining whether a protocol is syntactically correct and can be realized as an actual protocol run. Verifiers also differ in their methods of expressing protocol goals. In some cases ([13]), the protocol designer specifies goals using a fully general goal language; this allows flexibility in goal statement. Typical goals include maintaining *secrecy* of a given piece of information from an eavesdropper and *authentication* of one participant to another. Additional goals, such as *non-origination* (a participant never transmits a given atom or text) and *uniqueness* of a value, may be included by some verifiers. CPSA, for example, specifies goals thusly:

(deflistener m)
(non-orig (privk b))

The first example specifies that the value *m* should not be overheard by an intruder; the second specifies that the role being evaluated in this instance should never transmit the private key of *B*.

Other tools, such as Athena [21], predefine the goals available to the designer. Athena provides specific goals for authentication, secrecy, session key freshness, and session key verification (one participant verifies that another participant has also received the session key). If the analyst wishes to test a goal not already provided, she must alter the source code of the tool.

Since goal specification and availability differ from verifier to verifier, our generator does not specify goals as part of a protocol. Insertion of security goals when either required or desired should occur during translation of a generated protocol into the input language of the chosen verifier.

3.2.5 Finite vs. Infinite Search Space

Protocol verifiers vary both in the size of the search space allowed and in requirements for explicit user specification of that space. Casper [31] limits the search space to a finite number of protocol instances. These instances must be explicitly listed in the protocol specification, as in this excerpt from the simplified Needham-Schroeder public key protocol:

$INITIATOR(Alice, N_a)$
$RESPONDER(Bob, N_b)$

In this example, Casper creates one instance of the *INITIATOR* role (performed by Alice) and one instance of the *RESPONDER* role (performed by Bob).

ProVerif [30] also requires that the protocol analyst explicitly specify the search space, but allows specification of an infinite number of role instances. This is shown in the the ProVerif specification of the same protocol:

((!processA) | (!processB))

This statement explicitly specifies an infinite number of instances of *A* as initiator and *B* as responder.

Systems such as CPSA [13] and Athena [21] automatically search an infinite number of instances of any role without requiring user specification.

While implicit specification of a finite search space is possible, none of the verifiers surveyed took this approach.

Specification of search space is a function of protocol analysis rather than generation. Our generator, therefore, does not provide any representation of the search space. As protocols from the generator require translation into the appropriate input language for any given verifier, we leave specification of search space for implementation in the translator modules as appropriate.

3.2.6 Summary

We conclude this chapter by summarizing our design decisions below.

Area of Disagreement	*Decision*
Analysis procedure	Verifier specific; no impact on protocol generation
Independent processes v. intended run	Independent processes used in naive generation; intended run used in realistic generation
Implicit v. explicit deconstruction	Implicit (pattern matching)
Treatment of goals	Verifier specific; no impact on protocol generation
Finite v. infinite search space	Verifier specific; no impact on protocol generation

Table 3.1: Design decisions

With the common protocol definition of Figure 3.2 and the refinements on protocol generation rising from these decisions, we may now examine the problem of random generation over the protocol space.

CHAPTER 4

Random Protocol Generation

In this chapter, we define multiple probability distributions over the set of security protocols. We first examine our definition of a security protocol to identify the degrees of freedom possible for random selection of protocol elements. We then propose probability distributions over these degrees of freedom for generation of protocols ranging from the naive to the realistic.

4.1 Protocol Degrees of Freedom

Protocol generation may incorporate random selection at all levels. Each of these potential locations for random selection constitutes a *degree of freedom* in the protocol definition. To begin, we evaluate the common definition of a protocol found in Figure 3.2 to identify both these degrees of freedom and the available selections for each degree. The results of this evaluation are summarized below:

- *protocol*: number of roles; number of message nodes in the protocol
- *role*: number of messages nodes associated with each role
- *message node*: in the context of a given role, messages may be either sent or received

- *message body*: selection of message contents (encryption, concatenation, or atom)
- *atom*: names of participants, nonces/timestamps, or keys
- *concatenation*: concatenation of two components, two messages, or a component and a nested message
- *encryption*: selection of encryption key
- *keys*: selection of a symmetric or an asymmetric key
- *asymmetric*: public or private key used/said
- *symmetric*: long-term or session key used/said
- *message depth*: depth of the most nested atom of a message

We also note that any given message may contain both freshly created components (*e.g.,* generation of a nonce) and previously existing components (*e.g.,* repetition of that nonce).

4.2 Naive Probability Distributions

Next, we describe three naive probability distributions over these random factors, and discuss their advantages and disadvantages. We consider these distributions "naive" because they create protocols with little or no reference to the characteristics of real world protocols.

First, we describe an unbounded naive distribution, which creates protocols with the greatest possible degree of randomness. Second, we describe two modified distributions, which limit the randomness of the generator. The bounded-naive distribution limits the parameters and recursion in a generated protocol, while the intended-run naive distribution creates protocols as sequences of random send/receive message pairs between protocol participants.

4.2.1 Unbounded naive distribution

In the simplest distribution, the generator randomly selects protocol components at all levels with uniform probability. To accomplish this, we use the most general possibilities as described below:

- The number of roles in a protocol may be any positive integer.
- The number of message nodes in each role may be any positive integer; the number of message nodes in a protocol is the sum of the message nodes in the protocol's roles.
- Message direction is an even probability in every instance. This may cause unusual behavior when compared to real-world protocols. For instance, a role might only send messages, or only receive them; the messages it expects may never be sent by another role in the protocol, and messages it sends may likewise never be expected by another role.
- Each message is equally likely to be an atom, a concatenation of two messages, or an encryption of another message.
- An atom is equally likely to be a name, nonce, or key. Atoms are equally likely to either already exist in the protocol or be created freshly for that message.
- The first and second items of a concatenation are each equally likely to be an atom or a message.
- Keys may be either symmetric or asymmetric. Symmetric keys may be either long-term keys or session keys; asymmetric keys may be either public or private. In particular, any key may be said or used at any time, and a participant may send keys in the clear.

Given these, we may now specify algorithms for random protocol and message generation. At the protocol, role, and node levels, the generator chooses the quantity of the next lower element, then fills each lower element in turn. This algorithm is found in Figure 4.1.

At the protocol level, the generator selects the number of roles in the protocol (line 1), then uses the role-level random method to fill each role (line 2). The generator behaves similarly at the role level, selecting the number of nodes for each role and calling on the node-level random method.

To choose the message sent or received for each node in line 6 of the algorithm, the generator makes use of the message selection algorithm found in Figure 4.2.

```
 1: select number of roles in the protocol
 2: for each role do
 3:     select number of nodes in this role
 4:     for each node do
 5:         select direction (send or receive)
 6:         select message using message creation algorithm
 7:     end for
 8: end for
```

Figure 4.1: Protocol creation algorithm for unbounded distribution

```
 1: select reuse of an existing message (if any) or generation of a new message
 2: if selection is reuse of an existing message then
 3:     select message from list of available messages
 4: else
 5:     select type of message: encryption, concatenation or atom
 6:     if selection is encryption then
 7:         select type of key: public key,session key,or long-term-pair key
 8:         select reuse of an existing key or generation of a new key of the selected type
 9:         if selection is reuse of an existing key then
10:             select message from list of available key of the selected type
11:         else
12:             create a new key of the selected type and insert into message
13:         end if
14:         select encrypted message (recursive call)
15:     else if selection is concatenation then
16:         select first message of concatenation (recursive call)
17:         select second message of concatenation (recursive call)
18:     else
19:         select type of atom: name, nonce, or key
20:         select reuse of an existing atom or generation of a new atom of the selected
            type
21:         if selection is reuse of an existing atom then
22:             select message from list of available atoms of the selected type
23:         else
24:             create a new atom of the selected type and insert into message
25:         end if
26:     end if
27: end if
```

Figure 4.2: Message creation algorithm for unbounded naive distribution

This method of protocol generation provides the maximum exercise of the protocol degrees of freedom, and therefore has the potential to provide the most thorough coverage of atypical cases when used to generate test material. There are, however, notable disadvantages to this level of randomness. First, generation based on this probability distribution is highly likely to produce nonfunctional protocols (*i.e.,* protocols that cannot actually be executed in their entirety by all participants). Such protocols are useful for detecting gross flaws in protocols, but do not contribute to testing for the detection of subtle protocol flaws. Moreover, the unlimited recursion in this method is patently problematic.

4.2.2 Limited naive distributions

The extreme behavior of the unbounded naive distribution is clearly of limited utility. We therefore consider methods of limiting protocol generation while preserving the uniform nature of random factor selection in the generation process.

Intended-run naive distribution

The intended-run naive distribution is a lightly modified version of the unbounded naive distribution. For this distribution, we leave message selection unaltered. However, we incorporate the realistic assumption that security protocols are meant to be an exchange of messages between two or more participants. Accordingly, we modify the unbounded naive distribution in the following three ways:

- A protocol contains at least two roles.
- The number of message nodes in each protocol may be any positive integer; the number of message nodes in each role may be any positive integer less than or equal to the number of message nodes per protocol.
- Each message node will appear as a sent message in one role and as a received message in another. This must result in a partial ordering of the protocol messages, so that all roles may agree on the order of communication.

To accomplish this, we modify the selection of the number of roles to require a positive integer greater than or equal to two. Next, we add an explicit selection at the protocol level for the total number of nodes in the protocol and remove the selection of nodes per role. We then alter the protocol selection algorithm thus: as each node is generated, the generator randomly selects two roles, then adds the node to one role as a sent message and to the other role as a received message. After all messages have been distributed, we verify that all roles contain at least one message, and add a message with a random correspondent and direction to those roles that are empty. Random factors at the message level and below are selected in the same manner as in the unbounded naive message creation algorithm listed in Figure 4.2.

Protocol generation using this probability distribution will be more likely than the preceeding distributions to produce protocols that may be successfully completed by all participants. This is not guaranteed, as the naively-created protocols do not take into account the information actually available to each participant when composing messages. Since each message is sent by one participant and received by another, however, completion or noncompletion is dependent only on the ability of the participants to produce the specified messages. As this distribution does not place bounds on the number of protocol components, the generator is still subject to unlimited recursion.

```
1: select number of roles in the protocol
2: select number of nodes in the protocol
3: for each node do
4:    select message using naive message creation algorithm (recursive)
5:    Select sending role and insert node
6:    select receiving role and insert node
7: end for
```

Figure 4.3: Protocol creation algorithm for intended-run distributions

Bounded naive distribution

As before, the bounded naive distribution is based on the unbounded naive distribution. For this distribution, we leave the recursion in place, but introduce upper bounds on the number of elements in the protocol. We may easily limit the number of roles, message nodes per role, and atoms to a large but finite number. We also impose an upper bound on the degree to which components of a node's message may be nested (message depth). All other measures of randomness remain the same (for instance, a message sent by one role still may or may not ever be received by another). Accordingly, we modify the assumptions of the unbounded naive distribution as follows:

- A protocol may have a maximum of 10 roles.
- Each role may have a maximum of 10 message nodes. The implicit maximum number of message nodes in a protocol is the sum of the message nodes in the protocol roles.

- Messages may nest to a maximum depth of 10 levels.
- An atom is equally likely to be a name, nonce, or key. Atoms may already exist in terms of the protocol or be created freshly for that message, with a maximum of 100 for each type of atom.

To generate protocols according to this distribution, we use the protocol and message selection algorithms found in Figure 4.1 and Figure 4.2. We trivially modify the algorithms to incorporate the upper bounds checks on roles, nodes, atoms, and message depth.

The bounded naive distribution maintains maximum random selection within the distribution bounds, which in turn curbs the unlimited recursion of the unbounded distribution. However, the other problem with the unbounded naive distribution remains, as the bounded naive distribution is no more likely to produce functional protocols. To address this problem, we need to consider other limits on the naive distribution.

4.2.3 Limits of naive generation

While we have resolved the most egregious problems associated with naive protocol generation, there are several reasons why naive generation may be undesirable.

First, since random selection in the naive distribution (within either specified or computational bounds) is made with uniform probability, these distributions do not statistically resemble the real-world protocols that protocol verifiers are designed to test. Improvement in the test cases will lead to improved testing of the verifiers themselves.

Second, these distributions likewise incorporate no notion of desired protocol properties. The errors they contain are of the sort likely to be spotted by manual analysis. The main benefit of automatic protocol verification is discovery of subtle protocol flaws. We therefore wish to increase the likelihood of generating protocols whose flaws are subtle rather than gross, with the goal of improving verifiers' flaw detection capabilities.

Finally, not all protocol verifiers are capable of dealing with the overly large protocols frequently generated using the naive distribution. In the case of verifiers capable of dealing with infinite search space, an overly large protocol may mean failure to terminate, which is highly undesireable behavior.

4.3 Realistic Probability Distribution

For these reasons, we desire the capacity for *realistic* random generation, which produces protocols that statistically resemble existing protocols in structure and content. To accomplish this requires substantial narrowing of the degrees of freedom used in protocol generation.

In order to determine the properties of realistic protocols, we naturally begin by examining actual extant protocols. For this purpose, we draw a sample from the Clark-Jacob survey of authentication protocols [12]. Of the forty protocols in the Clark-Jacob library, thirty-two contain only those protocol elements found in the common protocol definition of Figure 3.2. We select only these protocols for analysis. By doing so, we eliminate the need to adjust for the presence of extraneous elements.

Our analysis of the sample is two-pronged. First, we identify any general properties of realistic protocols that we may incorporate as constraints into the generator. By requiring that these constraints be satisfied, we remove certain random factors as a consequence. Second, we statistically measure the components and construction of the sample protocols so that we may replicate those qualities when generating protocols.

4.3.1 General properties of actual protocols

Examination of the protocol sample yields these common properties of realistic protocols:

- A protocol has at least two participants.
- A protocol has an intended run; *i.e.,* each message of the protocol is sent by one protocol participant and received by another. No sender transmits a message to itself.
- Each participant sends or receives at least one message.
- Names may be reliably mapped to their keys.
- Each participant has either only one public/private key pair or only one long-term shared key.
- No participant in a protocol sends private or long-term shared keys in the clear.

- All names appearing in messages of a protocol are names of purported participants in the protocol run.

- Each encryption is accomplished using a key accessible to one of the participants. Note that this does not necessarily mean the *first* recipient of that message. Several protocols involve one participant sending information to another via an intermediate participant who cannot read it. We refer to an unreadable message segment meant to be forwarded to a third party as an *opaque term*.

The first and second properties have been previously incorporated as part of the intended-run naive distribution (Section 4.2.2), which limited the protocol, role and node selection while maintaining naive selection in the creation of messages. The other properties listed above describe constraints on messages and their contents. By using these constraints as assumptions for protocol generation, we narrow the scope of generation to protocols reflecting these known desired properties.

4.3.2 Statistical analysis of existing protocols

In addition to the new constraints introduced above, we choose a more detailed subset of the protocol degrees of freedom listed in Section 4.1 for statistical analysis:

- Number of roles in a protocol
- Number of messages in a protocol
- Number of unique atoms in a protocol (collectively and by type)
- Number of messages sent by a role
- Number of messages received by a role
- Number of unique atoms in a message (collectively and by type)
- Total number of atoms in a message
- Number of encryptions in a message
- Number of opaque terms in a message
- Message depth

For this analysis, we examine some of these factors in greater detail than was necessary for naive generation. While the number of roles and nodes per protocol provide sufficient detail as they were previously treated, we are now interested in both the number of nodes per role and in how many of those nodes represent messages sent vs. messages received. We also wish to identify not only the atoms used in a given message, but the relationship of those atoms to the set of unique atoms in the entire protocol. As the number of concatenations required to assemble a message is one less than the number of atoms in the message, we do not analyze concatenations. We do, however, add analysis of the opaque terms in a message.

Detailed results of this evaluation may be found in Appendix A. We summarize our findings in Table 4.1 and Table 4.2 below.

Protocol and Role Statistics				
Random Measure	*Min*	*Max*	*Mean*	*Standard Deviation*
Roles/protocol	2	4	2.688	0.535
Messages/protocol	1	8	4.094	1.838
Messages sent/role	0	4	1.524	0.836
Messages received/role	0	5	1.524	0.904
Unique names/protocol	2	3	2.156	0.369
Unique nonces/protocol	1	6	2.469	1.319
Unique keys/protocol	1	4	0.813	0.738
Unique opaque terms/protocol	0	2	0.719	0.683
Unique atoms/protocol	3	12	6.156	2.398

Table 4.1: Protocol and role statistics from the Clark-Jacob protocol library sample

Message Statistics				
Random Measure	*Min*	*Max*	*Mean*	*Standard Deviation*
Message depth	1	9	3.718	1.962
Unique atoms per message	1	7	2.870	1.541
Unique names per message	0	3	1.115	0.874
Unique nonces per message	0	3	1.305	0.722
Unique keys per message	0	2	0.254	0.471
Unique opaque terms per message	0	1	0.198	0.400
Total atoms per message	1	11	3.313	2.253
Encryptions per message	0	2	0.817	0.688

Table 4.2: Message statistics from the Clark-Jacob protocol library sample

In all cases, the analysis results from the Clark-Jacob library are significantly lower than the upper bounds used in the bounded naive distribution in Section 4.2.2. We particularly note the number of roles per protocol, the number of unique atoms per protocol, and the total number of atoms per message as remarkably more constrained than under the naive distributions. All the degrees of freedom studied, however, are suitable for stronger protocol generation limits.

Range of variables
With this analysis, we no longer need to rely on termination of recursion or coded upper bounds in the protocol and message generation process. Instead, we use our results to generate protocol components in more realistic quantities. For ease of implementation, we base this distribution on the standard normal (Gaussian) distribution; future realistic distributions would benefit from the application of asymmetric random generation, should this be available.

For each degree of freedom, we first generate a z-score according to a standard normal (Gaussian) distribution. We then multiply the z-score by the standard deviation for that degree of freedom and add the result to the appropriate mean. Finally, we round the result to the nearest integer. Should this result in a nonsensical value (*e.g.,* a negative value or a protocol with zero or one roles), we discard the result and repeat the process. Component selection according to this process produces results in accordance with a truncated Gaussian distribution.

4.3.3 Realistic selection algorithm

The generation algorithm for realistic distributions differs significantly from those of the previous distributions in placement of the random selection process. In the naive distributions, each protocol level for the most part selected only the components of the next level in the hierarchy, relying on that level to fill itself recursively. (The only exception to this is the intended-run naive distribution, in which selection of both roles of the protocol and nodes for each role occurs at the protocol level.) In particular, the components at the message level and below are capable of generating new atoms and operations during the message creation process. The number of atoms and operations in the entire protocol, therefore, is only known if and when the recursive generation process terminates.

In the realistic distribution algorithm, all atoms for the protocol are created before any messages are generated, and the construction set of atoms and operations in each message are selected before its composition, as shown in the realistic protocol generation algorithm of Figure 4.4.

```
1: select protocol and role parameters, including number of messages in protocol,
   number of atoms, etc.
2: build source set of names (all participant names)
3: build source set of keys (all public and long-term-shared keys, generate selected
   number of session keys)
4: build source set of nonces (generate selected number)
5: combine above source sets into master atom set $A$ for the protocol
6: **for** each message **do**
7:    select a sending and receiving role
8:    select total number of atoms $T$ for that message
9:    select unique atoms for that message from set of protocol atoms
10:   add $T$ atoms to the building-set $B$ for this message
11:   add $T$-1 concatenation operations to the building-set $B$
12:   select number of encryptions to add to building-set $B$
13:   **while** $B$ is not empty **do**
14:      select atom, contentation or encryption, removing from $B$
15:      **if** selection is concatenation **then**
16:         select a message for each side of concat as message
17:      **else if** selection is encryption **then**
18:         select encrypting key from available keys and
19:         select contents of encryption
20:      **else**
21:         insert atom into message
22:      **end if**
23:   **end while**
24: **end for**
```

Figure 4.4: Protocol and message creation algorithm for realistic distribution

As in the intended-run naive distribution, we begin by selecting the number of roles and nodes in the protocol. At this point, we also select the overall number of message components in the protocol. Second, we build source sets from which those message components may be drawn. These source sets include the names of all roles in the protocol and the public keys or long-term shared keys associated with those roles. Third, we create the actual building-set for the protocol by selecting the chosen number of names and keys from the source sets and generating the specified number of session keys and nonces.

For each message, the generator selects a sending and receiving role as in the intended-run naive distribution. It then creates the building-set of components for the message as follows. First, the generator selects the properties of the message; this includes the number of unique atoms in the message, the total number of atoms in the message, and the number of encryptions. It then chooses the correct number of atoms from the protocol building-set and adds these atoms to the message building-set, creating duplicates as necessary to provide the selected total number of atoms. Next, the generator adds the specified number of encryptions and enough concatenations to join the selected atoms. Once the building-set for the message has been created, the algorithm builds the message by removing items from the set, filling encryption and concatenation operations as appropriate, until the building-set is empty.

As the set of available message components was generated at the outset, we do not need to directly limit the depth of nesting in messages, as it is implicitly limited by the size of that set. This also enforces termination of the recursive message generation process when the supply of available atoms and operations has run out. Likewise, since the algorithm may select any encrypting key, we need not specifically direct the generator to create opaque terms.

This distribution does not create protocols so flawed as to be nonfunctional; if the protocol participants have the ability to produce the specified messages, the protocol run will complete. This may cause problems with verifiers which do not check for protocol completion as a component of correctness. However, the realistic distribution is more satisfactory for several reasons. First, as noted above, the recursion existing in the naive distributions is no longer a factor; the message creation process will always terminate. Second, the protocols generated using this distribution bear a much stronger resemblance in scope to real-world protocols, and will therefore work with a greater number of verifiers. Third, the statistical resemblance of these protocols to real-world protocols provides a superior set of test cases for those hoping to improve the quality of protocol flaw detection.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5

Implementation and Results

To validate our design, we implemented a generator protocol in Java that uses the distributions discussed in Chapter 4 to create random protocols. The initial implementation of the generator included random selection functions for the unbounded naive and realistic (Gaussian) distributions. Implementation of these distributions, while straightforward, required alterations in some cases from the original design.

5.1 Implementation of Distributions

5.1.1 Unbounded Naive

True unbounded naive generation, as it turns out, is generally impractical. Protocols generated naively may be an arbitrarily large size, limited only by computational resources and the size of the numeric data type used in the generator implementation. It is therefore not only possible for the generator to overrun the available stack space during the generation process; in the case of unbounded generation, it is highly likely. Also, it is not evident that protocols of such extreme size are useful for more than stress-testing protocol verifiers.

To test the general efficiency of the distribution, we first limit the number of roles and of nodes per role to one hundred (twenty times that of the largest protocol surveyed from the Clark-Jacob library), and the nesting depth of messages to twenty (more than twice the depth of the most-nested message in the Clark-Jacob protocol library). Even with these limits, unbounded naive generation fares poorly; in 10,000 attempts, the generator produced only 594 protocols, and required 3124.33 seconds (just over 52 minutes) to do so. The remaining 9,406 attempts resulted in stack overflow errors, which aborted the generation process.

We further limited the experiment to twenty roles, twenty nodes per role, and a message nesting depth of fifteen. Under these limits, the generator produced an average of 2678 protocols in 10,000 attempts, requiring an average of 1881.44 seconds (just under 31.5 minutes).

While this represents a significant improvement, we would prefer a significantly higher success rate for the generator. In order to more reliably produce protocols while retaining maximum randomness, we implement two limited naive distributions, and discuss the results of these distributions below.

5.1.2 Intended-run Naive

The intended-run naive distribution limits generation by producing protocols in which all messages were sent by one participant and received by another, but did not place any restrictions on message creation. We tested this distribution using the sets of limits previously used for the unbounded naive distribution. With the 100/100/20 limits, the generator produced an average of 3159 protocols out of 10,000 attempts, requiring 1930.888 seconds (just over 32 minutes) to do so. Using the more strict 20/20/15 limits, the successful generation rate improved to 4719 protocols in 10,000 attempts, with an average generation time of 1688.0967 (just over 28 minutes).

The results of the unbounded naive and the intended run-naive distributions are summarized for convenient comparison in Table 5.1.

5.1.3 Bounded Naive

The bounded naive distribution limits generation by placing upper bounds on several of the protocol degrees of freedom until the new distribution function runs with only minimal stack overflow errors. These bounds fall into two categories: those describing higher-level parameters (roles, nodes, message depth) as used with the previous naive distributions, and bounds on the number of message atoms created during protocol generation.

Unbounded and Intended-Run Distribution Results					
Distribution	*Roles/ Protocol*	*Nodes/Role*	*Message Depth*	*Successes in 10K Attempts*	*Average Run Time (seconds)*
Unbounded	100	100	20	594	3124.330
Intended-run	100	100	20	3159	1930.888
Unbounded	20	20	15	2678	1881.44
Intended-run	20	20	15	4719	1688.0967

Table 5.1: Results of unbounded and intended-run distribution testing

The bounded naive distribution easily outperforms both the unbounded and the intended-run distributions, attaining a generation success rate of ninety-six percent or higher with minimal bounding. Further restrictions, however, yield diminishing returns, as significant tightening of the upper bounds leads to less than two percent improvement in the protocol generation success rate. We discuss the effects of tightening each of the two bounds categories below.

Bounds on roles, nodes, and message depth

Table 5.2 lists the results of manipulating the bounds on the number of roles per protocol, the number of nodes per role, and the nesting depth of each message.

Role, Node and Message Depth Bounds (Atomic generation bounded at twenty instances of each)				
Roles/Protocol	*Nodes/Role*	*Message Depth*	*Successes in 10K Attempts*	*Average Run Time (seconds)*
20	20	15	9709	1540.555
10	10	10	9773	1377.290
5	10	10	9796	1374.693
10	5	10	9798	1376.834
10	10	5	9765	1378.469
5	5	10	9852	1374.722
5	10	5	9816	1373.732
10	5	5	9828	1376.008
5	5	5	9841	1374.759

Table 5.2: Results of role, node and message depth bounds manipulation

As the data show, tightening each of these bounds either individually or in combination results in a less than two percent improvement in successful protocol generation.

Within that range, lowering the upper bounds on roles and nodes has the most affect on successful generation, while lowering the upper bound on message depth is less effective. Generation time in all cases but the loosest fall within five seconds of each other. Given this, we recommend setting these bounds at ten for each parameter above. This allows a fifty percent increase in these random factors with the loss of only one percent of protocol generation success.

Bounds on atom creation

Table 5.3 lists the results of manipulating the bound on generation of each type of atom.

Atomic Bound Manipulation Results					
Maximum Number of Atoms	*Roles/ Protocol*	*Nodes/Role*	*Message Depth*	*Successes in 10K Attempts*	*Average Run Time (seconds)*
100	20	20	15	9707	1523.819
50	20	20	15	9718	1529.791
20	20	20	15	9709	1540.555
10	20	20	15	9695	1511.944
100	5	5	5	9829	1383.120
50	5	5	5	9835	1385.381
20	5	5	5	9841	1374.759
10	5	5	5	9828	1386.931

Table 5.3: Results of various bounds on atomic components

Clearly, decreasing the number of atoms generated has little effect on the success rate in comparison to tightening the upper-level bounds discussed in Section 5.1.3. We therefore set our atom limit at 100, which allows us to retain maximum generation without significantly altering the success rate.

5.1.4 Realistic

Implementation of the realistic distribution, by contrast, was quite straightforward. Protocol generation using this method was successful in every case, generating 10,000 protocols in an average of 1366.324 seconds.

5.2 Validation of Generated Protocols

To validate our protocol generation work against a protocol verifier, we created a translation interface between the output language of our generator and the CPSA input language. This required slight modifications to the intended-run distribution. As originally designed and implemented, this distribution could produce roles that did not transmit or receive any messages. CPSA considers a protocol with this type of role to be malformed. We therefore altered the affected distribution to include at least one message node in every role. CPSA could successfully parse all protocols produced using each of the four probability distributions. We did not have CPSA analyze these protocols for correctness. Sample protocols may be found in Appendix B.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 6

Future Work

In this chapter, we suggest areas for further research.

In order to increase the generator's utility as a means of comparing protocol verifiers, we see the need for creation of additional translation interfaces for verifier input languages, as was implemented in this work for CPSA. Priority should be given to input languages for those verifiers most commonly in use and to the common protocol languages (HLPSL and CAPSL).

Additional probability distributions on the set of protocols may yield more focused results while still retaining a suitable degree of randomness. For example, simplicity is often a desirable property of a protocol. Creation of probability distributions based on any of the asymmetric distribution functions would yield a random selection of protocols that tend to contain less elements. It may also be interesting to combine different distributions for any of the degrees of freedom in the protocol definition.

Many protocol verifiers require the specification of both a protocol and a goal for meaningful analysis. Unfortunately, as noted in Section 3.2.4, few verifiers specify goals in the same way. For this reason, actual selection of goals for a protocol specification is necessarily left to the translation interface for each verifier. Further research could yield generator improvements that simplify the translation process. This increases the utility of the generator as a supply of test cases for each verifier. It is, however, insufficient as a means of providing a set of verifier comparison tests. Direct comparison of verifiers requires that they test identical goals. Provision of this functionality would involve the creation of a common definition for goals (analogous to that developed in this work for protocols) and implementation of exhaustive and/or random goal generation.

APPENDIX A

Appendix A–Detailed Analysis of the Clark-Jacob Protocol Library

A.1 Protocol and Role Analysis

A.1.1 Protocol and role details

Protocol and Role Statistics				
Protocol	*Roles/protocol*	*Msgs/protocol*	*Messages sent/role*	*Messages re-ceived/role*
ISOSK1PU	2	1	0	1
			1	0
ISOSK2PU	2	2	1	1
			1	1
ISOSK2PM	2	2	1	1
			1	1
ISOSK3PM	2	3	2	1
			1	2
NRF	2	3	2	1
			1	2

AndrewRPC	2	4	2	2
			2	2
ISO1PUCCF	2	1	1	0
			0	1
ISO2PUCCF	2	2	1	1
			1	1
ISO2PMCCF	2	2	1	1
			1	1
ISO3PMCCF	2	3	2	1
			1	2
NSSK	3	5	3	2
			1	1
			1	2
DenningSacco	3	3	2	1
			1	1
			0	1
OtwayRees	3	4	1	1
			1	1
			2	2
ANSSK	3	7	4	3
			2	3
			1	1
WideMouthedFrog	3	2	1	0
			1	1
			0	1
Yahalom	3	4	2	1
			1	2
			1	1
Carlsen	3	5	2	1
			2	3
			1	1
ISO4P	3	4	2	2
			1	1

			1	1
ISO5P	3	5	2	1
			2	3
			1	1
WooLamPif	3	5	2	1
			2	3
			1	1
WooLamPi	3	5	2	1
			2	3
			1	1
WooLamMutual	3	7	3	2
			3	4
			1	1
KerberosV5	4	6	3	3
			1	1
			1	1
			1	1
NeumanStubblebineI	3	4	2	1
			1	2
			1	1
NeumanStubblebineII	3	7	4	2
			2	4
			1	1
KLSI	3	5	2	1
			2	3
			1	1
KLSII	3	8	4	2
			3	5
			1	1
KaoChow1	3	4	2	1
			1	2
			1	1
KaoChow2	3	4	2	1

			1	2
			1	1
KaoChow3	3	4	2	1
			1	2
			1	1
BilateralKEPK	2	3	2	1
			1	2
NSPK	3	7	3	2
			2	3
			2	2

A.1.2 Protocol-level atomic composition

Protocol-level Atomic Composition					
Protocol	*Unique names/ protocol*	*Unique nonces/ protocol*	*Unique keys/ protocol*	*Unique opaque terms/ protocol*	*Unique atoms/ protocol*
ISOSK1PU	2	1	0	0	3
ISOSK2PU	2	1	0	0	3
ISOSK2PM	2	2	0	0	4
ISOSK3PM	2	2	0	0	4
NRF	2	2	1	0	5
AndrewRPC	2	3	1	0	6
ISO1PUCCF	2	1	0	0	3
ISO2PUCCF	2	1	0	0	3
ISO2PMCCF	2	2	0	0	4
ISO3PMCCF	2	2	0	0	4
NSSK	2	2	1	1	6
DenningSacco	2	1	2	1	6
OtwayRees	2	3	2	2	9
ANSSK	2	3	1	2	8
WideMouthedFrog	2	2	1	0	5
Yahalom	2	2	1	1	6

Carlsen	2	3	1	1	7
ISO4P	2	4	1	1	8
ISO5P	2	3	1	1	7
WooLamPif	2	1	0	1	4
WooLamPi	2	1	0	1	4
WooLamMutual	2	2	1	2	7
KerberosV5	3	4	0	1	8
NeumanStubblebineI	2	4	1	1	8
NeumanStubblebineII	3	6	1	1	11
KLSI	2	4	2	1	9
KLSII	2	6	2	2	12
KaoChow1	3	2	1	1	7
KaoChow2	3	2	2	1	8
KaoChow3	3	3	2	1	9
BilateralKEPK	2	2	1	0	5
NSPK	2	2	0	0	4

A.2 Message Analysis

A.2.1 Message-level atomic composition

Message Atom Statistics					
Protocol	*Unique Names/ Msg*	*Unique Nonces/ Msg*	*Unique Keys/ Msg*	*Unique Opaque Terms/ Msg*	*Unique Atoms/ Msg*
ISOSK1PU	2	1	0	0	3
ISOSK2PU	1	1	0	0	2
	1	1	0	0	2
ISOSK2PM	2	1	0	0	3
	1	1	0	0	2
ISOSK3PM	1	1	0	0	2
	1	2	0	0	3
	0	2	0	0	2
NRF	1	1	0	0	2

	1	2	1	0	4
	1	1	0	0	2
AndrewRPC	1	1	0	0	2
	0	2	0	0	2
	0	1	0	0	1
	0	1	1	0	2
ISO1PUCCF	2	1	0	0	3
ISO2PUCCF	1	1	0	0	2
	1	1	0	0	2
ISO2PMCCF	2	1	0	0	3
	1	1	0	0	2
ISO3PMCCF	1	1	0	0	2
	1	2	0	0	3
	0	1	0	0	1
NSSK	2	1	0	0	3
	2	1	1	0	4
	0	0	0	1	1
	0	1	0	0	1
	0	1	0	0	1
DenningSacco	2	0	0	0	2
	2	1	1	0	4
	0	0	0	1	1
OtwayRees	2	2	0	0	4
	2	2	0	1	5
	0	2	1	0	3
	0	1	0	1	1
ANSSK	1	0	0	0	1
	1	1	0	0	2
	2	1	0	1	4
	2	2	1	0	5
	0	0	0	1	1
	0	1	0	0	1
	0	1	0	0	1

WideMouthedFrog	2	1	1	0	4
	1	1	1	0	3
Yahalom 1	1	0	0	0	2
	2	2	0	0	4
	2	2	1	0	5
	0	1	0	1	2
Carlsen	1	1	0	0	2
	2	2	0	0	4
	2	2	1	0	5
	0	2	0	1	3
	0	1	0	0	1
ISO4P	2	1	0	0	3
	2	2	1	0	5
	2	1	0	1	4
	1	1	0	0	2
ISO5P	1	1			2
	2	2	0	0	4
	2	2	1	0	5
	1	2	0	1	4
	0	2	0	0	2
WooLamPif	1				1
		1	0	0	1
	2	1	0	0	3
	2	1	0	1	4
	2	1	0	0	3
WooLamPi	1	0	0	0	1
		1	0	0	1
		1	0	0	1
	2	0	0	1	3
	0	1	0	0	1
WooLamMutual	1	1	0	0	2
	1	1	0	0	2
	2	2	0	0	4

	2	2	0	1	5
	2	2	1	0	5
	0	2	0	1	3
	0	1	0	0	1
KerberosV5	2	2	0	0	4
	2	2	1	0	5
	2	3	0	1	6
	3	2	1	0	6
	1	1	0	1	3
	0	1	0	0	1
NeumanStubblebineI	1	1	0	0	2
	2	3	0	0	5
	2	3	1	0	6
	0	1	0	1	2
NeumanStubblebineII	1	1	0	0	2
	2	3	0	0	5
	2	3	1	0	6
	0	1	0	1	2
	0	1	0	1	2
	0	2	0	0	2
	0	1	0	0	1
KLSI	1	1	0	0	2
	2	2	0	0	4
	2	2	1	0	5
	1	3	1	1	6
	0	1	0	0	1
KLSII	1	1			2
	2	2			4
	2	2	1		5
	1	3	1	1	6
		1	0	0	1
	1	1	0	1	3
	0	2	0	0	2

	0	1	0	0	1
KaoChow1	2	1	0	0	3
	3	1	1	0	5
	1	2	0	1	4
	0	1	0	0	1
KaoChow2	2	1	0	0	3
	3	1	2	0	6
	2	3	1	1	7
	0	1	1	0	2
KaoChow3	2	1	0	3	
	3	1	2	0	6
	2	3	1	1	7
	0	1	1	1	3
BilateralKEPK	1	1	0	0	2
	1	2	1	0	4
	0	1	0	0	1
NSPK	2	0	0	0	2
	1	0	1	0	2
	1	1	0	0	2
	2	0	0	0	2
	1		1	0	2
	1	2	0	0	3
		1	0	0	1

A.2.2 Message composition statistics

Message Composition Statistics				
Protocol	*Unique Atoms /Msg*	*Total Atoms /Msg*	*Encryptions /Msg*	*Msg Depth*
ISOSK1PU	3	3	1	6
ISOSK2PU	2	2	0	2
	2	2	1	3
ISOSK2PM	3	3	1	3
	2	2	1	3

ISOSK3PM	2	2	0	2
	3	3	1	4
	2	2	1	3
NRF	2	2	0	2
	4	5	1	8
	2	2	1	3
AndrewRPC	2	2	1	3
	2	2	1	3
	1	1	1	2
	2	2	1	3
ISO1PUCCF	3	5	0	6
ISO2PUCCF	2	2	0	2
	2	2	0	3
ISO2PMCCF	3	4	0	7
	2	3	0	4
ISO3PMCCF	2	2	0	2
	3	4	0	6
	1	2	0	3
NSSK	3	3	0	2
	4	5	2	7
	1	1	0	1
	1	1	1	2
	1	1	1	2
DenningSacco	2	2	0	2
	4	6	2	8
	1	1	0	1
OtwayRees	4	7	1	8
	5	8	1	9
	3	4	2	4
	2	2	0	2
ANSSK	1	1	0	1
	2	2	1	3
	4	4	0	5

	5	6	2	8
	1	1	0	1
	1	1	1	2
	1	1	1	2
WideMouthedFrog	4	4	1	5
	3	4	1	4
Yahalom	2	2	0	2
	4	4	1	5
	5	6	2	6
	2	2	1	3
Carlsen	2	2	0	2
	4	4	0	4
	5	6	2	5
	3	3	1	4
	1	1	1	2
ISO4P	3	3	0	2
	5	6	2	5
	4	4	1	5
	2	2	1	3
ISO5P	2	2	0	2
	4	4	0	4
	5	6	2	6
	4	4	1	5
	2	2	1	3
WooLamPif	1	1	0	1
	1	1	0	1
	3	3	1	4
	4	4	1	6
	1	1	1	4
WooLamPi	1	1	0	1
	1	1	0	1
	1	1	1	2
	3	3	1	4

	1	1	1	2
WooLamMutual	2	2	0	2
	2	2	0	2
	4	4	1	5
	5	7	1	8
	5	8	2	6
	3	3	1	4
	1	1	1	2
KerberosV5	4	4	0	4
	5	8	2	6
	6	6	1	7
	6	9	2	7
	3	3	1	4
	1	1	1	2
NeumanStubblebineI	2	2	0	2
	5	5	1	5
	6	8	2	6
	2	2	1	3
NeumanStubblebineII	2	2	0	2
	5	5	1	5
	6	8	2	6
	2	2	1	3
	2	2	0	2
	2	2	1	3
	1	1	1	2
KLSI	2	2	0	2
	4	4	0	4
	5	6	2	5
	6	6	2	5
	1	1	1	2
KLSII	2	2	0	2
	4	4	0	4
	5	6	2	5

	6	6	2	5
	1	1	1	2
	3	3	0	3
	2	2	1	3
	1	1	1	2
KaoChow1	3	3	0	3
	5	9	2	7
	4	4	1	4
	1	1	1	2
KaoChow2	3	3	0	3
	6	11	2	8
	5	5	1	5
	2	2	1	3
KaoChow3	3	3	0	3
	6	11	2	8
	7	9	2	9
	3	3	1	3
BilateralKEPK	2	3	1	3
	4	4	1	5
	1	1	1	3
NSPK	2	2	0	2
	2	2	1	3
	2	3	1	4
	2	2	0	2
	2	2	1	3
	3	3	1	4
	1	1	1	2

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B

Appendix B–Samples of Generated Protocols

B.1 Unbounded Naive Sample Protocol

This excerpt of a naively generated protocol contains one role of the sixty-nine generated for this protocol.

```
(defprotocol protocol1 basic
  (defrole role0
    (vars (name35 name)(gen_name86 name)(gen_name85 name)(name7 name)(name16 name)(name56 name)
           (name49 name)(gen_name87 name)(gen_name88 name)(gen_name89 name)(name48 name)
           (gen_name90 name)(name75 name)(name22 name)(name40 name)(gen_name91 name)
           (gen_name92 name)(name27 name)(gen_name93 name)(name73 name)(gen_name94 name)
           (gen_name95 name)(name47 name)(gen_name97 name)(name61 name)(gen_name96 name)
           (name9 name)(name34 name)(gen_name98 name)(gen_name99 name)(name62 name)(name55
           name)(gen_name100 name)(name77 name)(name79 name)(name80 name)(gen_name101 name)
           (name11 name)(name57 name)(name32 name)(gen_name102 name)(name81 name)(gen_name103
           name)(gen_name104 name)(name70 name)(name5 name)(name74 name)(name43 name)(name15
           name)(name66 name)(name52 name)(gen_name106 name)(gen_name107 name)(gen_name105
           name)(name60 name)(gen_name108 name)(gen_name109 name)(name19 name)(gen_name110
           name)(gen_name111 name)(gen_name112 name)(gen_name113 name)(name84 name)(name42
           name)(name1 name)(name45 name)(name65 name)(gen_name114 name)(name51 name)
           (gen_name115 name)(gen_name116 name)(name44 name)(gen_name117 name)(gen_name118
           name)(gen_name119 name)(name78 name)(name8 name)(name10 name)(name59 name)(name83
           name)(name50 name)(gen_name120 name)(name63 name)(gen_name121 name)(name36 name)
           (gen_name139 name)(gen_name122 name)(gen_name123 name)(gen_name124 name)(gen_name125
           name)(gen_name126 name)(name0 name)(gen_name127 name)(name46 name)(gen_name128
```

```
        name)(gen_name132 name)(gen_name131 name)(name82 name)(gen_name129 name)(gen_name130
        name)(gen_name134 name)(gen_name133 name)(gen_name135 name)(gen_name136 name)
        (gen_name137 name)(gen_name138 name)(name31 name)(gen_name140 name)(gen_name141
        name)(gen_name142 name)(name39 name)(gen_name145 name)(name4 name)(gen_name144
        name)(gen_name143 name)(gen_name146 name)(gen_name147 name)(gen_name148 name)
        (gen_name149 name)(gen_name150 name)(gen_name151 name)(gen_name152 name)
        (gen_name154 name)(gen_name153 name)(gen_name155 name)(gen_name156 name)
        (gen_name157 name)(gen_name158 name)(gen_name159 name)(name58 name)(gen_name160
        name)(nonce1 text)(nonce2 text)(nonce3 text)(nonce4 text)(nonce5 text)(nonce6
        text)(nonce7 text))
(trace
  (recv (enc (enc name7 (pubk gen_name85)) (ltk name35 gen_name86)))
  (recv (ltk name35 gen_name86))
  (send (ltk name16 name56))
  (recv nonce1)
  (recv name49)
  (recv nonce1)
  (recv (cat (cat (pubk gen_name87) gen_name88) (cat (enc name48 (pubk gen_name89)) (enc
        (enc (cat (enc gen_name87 (ltk gen_name90 name75)) (cat (cat (ltk gen_name91
        gen_name92) (enc name49 (ltk name27 gen_name93))) nonce1)) (ltk name22 name40))
        (ltk gen_name90 name75)))))
  (send (cat nonce1 (pubk gen_name89)))
  (recv (enc name16 (ltk name22 name40)))
  (send (pubk gen_name85))
  (recv name73)
  (recv (cat (enc gen_name94 (pubk gen_name89)) (ltk gen_name95 name47)))
  (recv (enc (cat (enc gen_name85 (ltk gen_name96 name9)) (enc nonce1 (ltk name27
        gen_name93))) (ltk gen_name97 name61)))
  (send (ltk name34 gen_name98))
  (send (cat (cat (ltk gen_name91 gen_name92) (enc name49 (ltk name27 gen_name93)))
        nonce1))
  (recv gen_name99)
  (send (enc name62 (ltk gen_name96 name9)))
  (send name55)
  (recv (cat (ltk gen_name100 name77) (enc (enc (enc name32 (ltk name11 name57)) (pubk
        gen_name101)) (ltk name79 name80))))
  (send (enc name62 (ltk gen_name96 name9)))
  (send (enc (ltk name9 gen_name102) (ltk name27 gen_name93)))
  (send (cat (cat (enc (ltk name81 name27) (ltk gen_name96 name9)) (enc (enc name70 (ltk
        gen_name103 gen_name104)) (pubk gen_name89))) (pubk gen_name85)))
  (recv name5)
  (recv (cat (enc name48 (pubk gen_name89)) (enc (enc (cat (enc gen_name87 (ltk
        gen_name90 name75)) (cat (cat (ltk gen_name91 gen_name92) (enc name49 (ltk
        name27 gen_name93))) nonce1)) (ltk name22 name40)) (ltk gen_name90 name75))))
  (send (cat (enc (enc name22 (pubk gen_name89)) (ltk name22 name40)) (enc name74 (ltk
        name9 gen_name102))))
  (recv (pubk gen_name85))
  (recv (enc name7 (pubk gen_name85)))
  (recv (ltk name35 gen_name86))
  (recv name43)
  (send gen_name90)
  (recv name15)
  (recv nonce2)
```

```
(recv name66)
(send name52)
(recv (cat nonce1 (cat (enc (cat (enc gen_name90 (ltk name81 name27)) gen_name105) (ltk
      gen_name106 gen_name107)) (cat (cat (pubk gen_name87) (enc (cat (cat gen_name108
      (cat (enc name19 (pubk gen_name109)) (ltk name16 name5))) (enc gen_name95 (pubk
      gen_name110))) (ltk gen_name99 name60))) gen_name111))))
(recv nonce3)
(send (enc gen_name87 (ltk gen_name90 name75)))
(send (cat (pubk gen_name87) gen_name88))
(recv name73)
(recv nonce3)
(send (cat (enc name5 (ltk gen_name112 gen_name113)) (enc name42 (ltk gen_name95
      name84))))
(recv name1)
(recv (pubk gen_name109))
(send (enc name48 (pubk gen_name89)))
(send gen_name112)
(send gen_name85)
(recv gen_name93)
(recv (enc nonce1 (ltk gen_name97 name61)))
(recv name45)
(send gen_name100)
(send name65)
(send (pubk gen_name114))
(recv name51)
(recv (enc (ltk gen_name106 gen_name107) (pubk gen_name101)))
(send (cat (cat (cat (enc name32 (ltk name16 name5)) (enc (enc name74 (ltk gen_name92
      gen_name89)) (ltk gen_name115 gen_name87))) (enc (pubk gen_name101) (ltk name27
      gen_name93))) (enc name48 (ltk gen_name111 name7))))
(send gen_name115)
(recv gen_name111)
(send (cat (cat nonce2 (cat (enc name47 (ltk gen_name116 name44)) (cat (cat (cat (enc
      (enc nonce1 (ltk name27 gen_name93)) (pubk gen_name117)) (enc name78 (ltk
      gen_name118 gen_name119))) (enc name8 (ltk gen_name100 name77))) nonce4))) name10))
(send name59)
(recv gen_name105)
(recv name83)
(recv name50)
(recv (enc (ltk name79 name80) (ltk gen_name120 name63)))
(send (cat (cat (enc gen_name110 (ltk gen_name103 gen_name104)) (enc (cat (ltk
      gen_name100 name77) (enc (enc (enc name32 (ltk name11 name57)) (pubk gen_name101))
      (ltk name79 name80))) (ltk gen_name121 name36))) (cat (cat (enc (ltk gen_name116
      name44) (pubk gen_name87)) (enc (cat (cat (cat (enc nonce1 (ltk name70
      gen_name122)) (cat (enc (enc gen_name103 (ltk gen_name123 gen_name124)) (ltk
      name45 gen_name106)) (enc gen_name120 (ltk gen_name125 gen_name126)))) name0) (cat
      (cat (cat (enc (pubk gen_name101) (pubk gen_name109)) (enc (cat (cat (cat (cat
      gen_name127 (ltk name81 name27)) name46) (pubk gen_name128)) (enc gen_name111
      (ltk gen_name95 name47))) (ltk gen_name120 name63))) (enc (enc (cat name82 (cat
      (cat (enc (cat (cat (ltk gen_name95 name84) gen_name129) (enc (pubk gen_name87)
      (ltk gen_name90 name75))) (ltk name16 name5)) (cat (ltk gen_name120 name63)
      nonce5)) (enc gen_name130 (ltk name79 name80)))) (ltk gen_name110 gen_name131))
      (ltk gen_name132 gen_name95))) (cat (enc (enc name42 (ltk gen_name95 name84)) (ltk
      name11 name57)) (cat (enc (cat name7 (cat (enc gen_name133 (pubk gen_name134))
      nonce5)) (ltk name22 name40)) (cat (cat (cat (enc gen_name125 (ltk gen_name135
      gen_name136)) gen_name137) (enc (enc (ltk name9 gen_name102) (ltk gen_name120
      name63)) (ltk name9 gen_name138))) (cat nonce6 nonce2)))))) (pubk gen_name139)))
      (enc gen_name112 (ltk name70 gen_name122)))))
(recv (cat (cat name31 (enc (enc (cat (cat gen_name108 (cat (enc name19 (pubk
      gen_name109)) (ltk name16 name5))) (enc gen_name95 (pubk gen_name110))) (ltk
      gen_name99 name60)) (ltk gen_name140 gen_name141))) (cat (ltk gen_name142 name39)
      (enc (enc (cat (enc (pubk gen_name143) (ltk name16 name5)) (pubk gen_name87)) (ltk
      name4 gen_name144)) (ltk name82 gen_name145)))))
(recv gen_name89)
(send gen_name146)
(send (cat gen_name147 gen_name148))
(send nonce1)
(send (enc gen_name137 (ltk gen_name149 name70)))
(send nonce4)
(recv (enc (pubk gen_name143) (ltk name4 gen_name144)))
(recv (enc (enc (cat (cat (enc gen_name110 (ltk gen_name103 gen_name104)) (enc (cat
      (ltk gen_name100 name77) (enc (enc (enc name32 (ltk name11 name57)) (pubk
      gen_name101)) (ltk name79 name80))) (ltk gen_name121 name36))) (cat (cat (enc
      (ltk gen_name116 name44) (pubk gen_name87)) (enc (cat (cat (cat (enc nonce1 (ltk
      name70 gen_name122)) (cat (enc (enc gen_name103 (ltk gen_name123 gen_name124))
      (ltk name45 gen_name106)) (enc gen_name120 (ltk gen_name125 gen_name126)))) name0)
      (cat (cat (cat (enc (pubk gen_name101) (pubk gen_name109)) (enc (cat (cat (cat
      (cat gen_name127 (ltk name81 name27)) name46) (pubk gen_name128)) (enc
      gen_name111 (ltk gen_name95 name47))) (ltk gen_name120 name63))) (enc (enc (cat
      name82 (cat (cat (enc (cat (cat (ltk gen_name95 name84) gen_name129) (enc (pubk
      gen_name87) (ltk gen_name90 name75))) (ltk name16 name5)) (cat (ltk gen_name120
      name63) nonce5)) (enc gen_name130 (ltk name79 name80)))) (ltk gen_name110
      gen_name131)) (ltk gen_name132 gen_name95))) (cat (enc (enc name42 (ltk gen_name95
      name84)) (ltk name11 name57)) (cat (enc (cat name7 (cat (enc gen_name133 (pubk
      gen_name134)) nonce5)) (ltk name22 name40)) (cat (cat (cat (enc gen_name125 (ltk
      gen_name135 gen_name136)) gen_name137) (enc (enc (ltk name9 gen_name102) (ltk
      gen_name120 name63)) (ltk name9 gen_name138))) (cat nonce6 nonce2)))))) (pubk
      gen_name139))) (enc gen_name112 (ltk name70 gen_name122)))) (ltk gen_name140
      name35)) (pubk gen_name150)))
(send gen_name104)
(recv (enc (cat nonce5 nonce6) (ltk gen_name151 gen_name152)))
(send gen_name124)
(recv (cat (enc (cat gen_name153 nonce7) (pubk gen_name154)) (cat nonce6 (ltk gen_name87
      gen_name155))))
(recv (cat (cat (enc (ltk gen_name116 name44) (pubk gen_name87)) (enc (cat (cat (cat
      (enc nonce1 (ltk name70 gen_name122)) (cat (enc (enc gen_name103 (ltk gen_name123
      gen_name124)) (ltk name45 gen_name106)) (enc gen_name120 (ltk gen_name125
      gen_name126)))) name0) (cat (cat (cat (enc (pubk gen_name101) (pubk gen_name109))
      (enc (cat (cat (cat (cat gen_name127 (ltk name81 name27)) name46) (pubk
      gen_name128)) (enc gen_name111 (ltk gen_name95 name47))) (ltk gen_name120 name63)))
      (enc (enc (cat name82 (cat (cat (enc (cat (cat (ltk gen_name95 name84) gen_name129)
      (enc (pubk gen_name87) (ltk gen_name90 name75))) (ltk name16 name5)) (cat (ltk
      gen_name120 name63) nonce5)) (enc gen_name130 (ltk name79 name80)))) (ltk
      gen_name110 gen_name131)) (ltk gen_name132 gen_name95))) (cat (enc (enc name42
      (ltk gen_name95 name84)) (ltk name11 name57)) (cat (enc (cat name7 (cat (enc
      gen_name133 (pubk gen_name134)) nonce5)) (ltk name22 name40)) (cat (cat (cat (enc
      gen_name125 (ltk gen_name135 gen_name136)) gen_name137) (enc (enc (ltk name9
          gen_name102) (ltk gen_name120 name63)) (ltk name9 gen_name138))) (cat nonce6
          nonce2)))))) (pubk gen_name139))) (enc gen_name112 (ltk name70 gen_name122))))
      (recv (cat (cat (enc nonce6 (pubk gen_name128)) (pubk gen_name156)) nonce5))
      (send (cat (ltk gen_name87 gen_name155) (enc (enc (ltk name9 gen_name102) (ltk
          gen_name132 name62)) (ltk gen_name132 name62))))
      (recv gen_name100)
      (recv (enc nonce1 (ltk name70 gen_name122)))
      (send (cat (enc (cat (cat gen_name157 (cat (enc name81 (ltk name79 name80)) (cat
          gen_name158 (ltk gen_name156 gen_name159)))) (cat name58 (enc (cat (ltk gen_name120
          name63) nonce5) (pubk gen_name160)))) (pubk gen_name89)) nonce1))
      (send (cat (cat (enc (cat (cat (ltk gen_name95 name84) gen_name129) (enc
          (pubk gen_name87) (ltk gen_name90 name75))) (ltk name16 name5)) (cat (ltk
          gen_name120 name63) nonce5)) (enc gen_name130 (ltk name79 name80))))
      (send (ltk name45 gen_name106))
      (send (enc (cat name82 (cat (cat (enc (cat (cat (ltk gen_name95 name84) gen_name129)
          (enc (pubk gen_name87) (ltk gen_name90 name75))) (ltk name16 name5)) (cat
          (ltk gen_name120 name63) nonce5)) (enc gen_name130 (ltk name79 name80)))) (ltk
          name9 gen_name138)))
      (send name81)
      (send (cat (enc nonce6 (pubk gen_name128)) (pubk gen_name156)))))
```

B.2 Intended-Run Naive Sample Protocol

```
(defprotocol protocol1 basic
  (defrole role0
    (vars (gen_name19 name)(gen_name20 name)(gen_name18 name)(gen_name21 name)(gen_name22 name)
           (gen_name9 name)(gen_name5 name)(gen_name6 name)(gen_name26 name)(gen_name7 name)
           (gen_name23 name)(gen_name24 name)(gen_name17 name)(gen_name25 name)(gen_name14
           name)(gen_name15 name)(gen_name16 name)(name0 name)(gen_name27 name)(nonce6 text)
           (nonce7 text))
    (trace
      (recv (cat (enc gen_name18 (ltk gen_name19 gen_name20)) (cat (enc (enc (enc (cat
          gen_name5 (pubk gen_name6)) (ltk gen_name21 gen_name22)) (pubk gen_name9)) (ltk
          gen_name21 gen_name22)) (enc (cat (cat (ltk gen_name23 gen_name24) (enc (enc
          gen_name17 (pubk gen_name6)) (pubk gen_name9))) (cat (cat (cat (cat (ltk gen_name25
          gen_name25) nonce6) gen_name26) nonce7) (enc (enc (ltk gen_name16 gen_name17)
          (pubk gen_name15)) (ltk gen_name5 gen_name14)))) (ltk gen_name26 gen_name7)))))
      (recv (ltk gen_name25 gen_name25))
      (recv name0)
      (send gen_name27)))
  (defrole role1
    (vars (nonce3 text)(nonce8 text))
    (trace
      (recv nonce3)
      (send nonce8)))
  (defrole role2
    (vars (gen_name5 name)(gen_name6 name)(gen_name7 name)(gen_name8 name)(gen_name9 name)
           (gen_name16 name)(gen_name17 name)(gen_name10 name)(gen_name12 name)(gen_name11
           name)(name4 name)(gen_name13 name)(gen_name15 name)(gen_name14 name)(name1 name)
           (gen_name27 name)(nonce1 text)(nonce2 text)(nonce3 text)(nonce4 text)
           (nonce5 text))
    (trace
      (send (cat gen_name5 (pubk gen_name6)))
      (send (cat (cat (cat (ltk gen_name7 gen_name8) (cat nonce1 (cat (enc (enc (ltk gen_name7
            gen_name8) (pubk gen_name6)) (pubk gen_name9)) (cat (enc (enc (cat (cat (cat
            nonce2 (enc (enc name4 (ltk gen_name10 gen_name11)) (ltk gen_name10 gen_name12)))
            (pubk gen_name13)) (cat (cat (cat nonce2 nonce3) nonce4) (enc (enc gen_name6 (ltk
            gen_name5 gen_name14)) (pubk gen_name15)))) (ltk gen_name16 gen_name17)) (ltk
            gen_name7 gen_name8)) name1)))) (ltk gen_name7 gen_name8)) nonce5))
      (send nonce3)
      (recv gen_name27)))
  (defrole role3
    (vars (gen_name5 name)(gen_name6 name)(nonce8 text))
    (trace
      (recv (cat gen_name5 (pubk gen_name6)))
      (recv nonce8)))
  (defrole role4
    (vars (gen_name7 name)(gen_name8 name)(gen_name9 name)(gen_name6 name)(gen_name16 name)
           (gen_name17 name)(gen_name10 name)(gen_name12 name)(gen_name11 name)(name4 name)
           (gen_name13 name)(gen_name15 name)(gen_name5 name)(gen_name14 name)(name1 name)
           (gen_name19 name)(gen_name20 name)(gen_name18 name)(gen_name21 name)(gen_name22
           name)(gen_name26 name)(gen_name23 name)(gen_name24 name)(gen_name25 name)(name0
           name)(nonce1 text)(nonce2 text)(nonce3 text)(nonce4 text)(nonce5 text)(nonce6 text)
           (nonce7 text))
    (trace
      (recv (cat (cat (cat (ltk gen_name7 gen_name8) (cat nonce1 (cat (enc (enc (ltk gen_name7
            gen_name8) (pubk gen_name6)) (pubk gen_name9)) (cat (enc (enc (cat (cat (cat
            nonce2 (enc (enc name4 (ltk gen_name10 gen_name11)) (ltk gen_name10 gen_name12)))
            (pubk gen_name13)) (cat (cat (cat nonce2 nonce3) nonce4) (enc (enc gen_name6
            (ltk gen_name5 gen_name14)) (pubk gen_name15)))) (ltk gen_name16 gen_name17)) (ltk
            gen_name7 gen_name8)) name1)))) (ltk gen_name7 gen_name8)) nonce5))
      (send (cat (enc gen_name18 (ltk gen_name19 gen_name20)) (cat (enc (enc (enc (cat
            gen_name5 (pubk gen_name6)) (ltk gen_name21 gen_name22)) (pubk gen_name9))
            (ltk gen_name21 gen_name22)) (enc (cat (cat (ltk gen_name23 gen_name24) (enc (enc
            gen_name17 (pubk gen_name6)) (pubk gen_name9))) (cat (cat (cat (cat (ltk
            gen_name25 gen_name25) nonce6) gen_name26) nonce7) (enc (enc (ltk gen_name16
            gen_name17) (pubk gen_name15)) (ltk gen_name5 gen_name14)))) (ltk gen_name26
            gen_name7)))))
      (send (ltk gen_name25 gen_name25))
      (send name0))))
```

B.3 Bounded Naive Sample Protocol

```
(defprotocol protocol1 basic
  (defrole role0
    (vars (gen_name2 name)(gen_name3 name)(name0 name)(gen_name4 name)(gen_name6 name)
           (nonce1 text)(nonce2 text))
    (trace
      (send (cat nonce1 (cat nonce1 (enc name0 (ltk gen_name2 gen_name3)))))
      (recv (ltk gen_name2 gen_name3))
      (send nonce2)
      (recv name0)
    (recv (enc (ltk gen_name2 gen_name4) (ltk gen_name2 gen_name3)))
    (send gen_name6)
    (recv gen_name4)
    (recv (cat nonce1 (cat nonce1 (enc name0 (ltk gen_name2 gen_name3)))))))
(defrole role1
  (vars (gen_name8 name)(gen_name11 name)(gen_name14 name)(gen_name12 name)(gen_name10 name)
         (gen_name5 name)(gen_name4 name)(name1 name)(gen_name13 name)(gen_name7 name)
         (gen_name2 name)(gen_name20 name)(gen_name21 name)(gen_name22 name)(gen_name16
         name)(gen_name23 name)(name0 name)(gen_name15 name)(gen_name3 name)(gen_name18
         name)(gen_name24 name)(nonce3 text)(nonce2 text)(nonce1 text)(nonce4 text)(nonce7
          text)(nonce8 text))
  (trace
    (send (cat gen_name8 nonce3))
    (recv (cat nonce2 nonce1))
    (send gen_name11)
    (recv nonce3)
    (recv (cat nonce4 nonce4))
    (send (pubk gen_name14))
    (recv (cat nonce3 (cat gen_name12 (pubk gen_name10))))
    (recv (cat (ltk gen_name5 gen_name4) (enc (ltk gen_name12 gen_name13)
          (ltk name1 gen_name8))))
    (send gen_name4)
    (recv (enc gen_name7 (ltk gen_name5 gen_name4)))
    (recv gen_name2)
    (send (cat nonce2 nonce1))
    (send (cat (ltk gen_name20 gen_name21) (cat (pubk gen_name22) gen_name16)))
    (recv (enc (ltk gen_name23 name0) (ltk gen_name7 gen_name22)))
    (recv (cat gen_name12 (pubk gen_name10)))
    (send (cat gen_name8 (cat gen_name14 (enc nonce7 (ltk gen_name15 gen_name3)))))
    (recv gen_name23)
    (recv (enc nonce8 (ltk gen_name18 gen_name24)))
    (send (cat name1 nonce4))))
(defrole role2
  (vars (gen_name21 name)(gen_name2 name)(gen_name3 name)(name0 name)(gen_name18 name)
         (gen_name6 name)(gen_name8 name)(gen_name24 name)(gen_name25 name)(nonce9 text)
         (nonce1 text)(nonce10 text)(nonce3 text))
  (trace
    (recv nonce9)
    (recv gen_name21)
    (send (cat nonce1 (enc name0 (ltk gen_name2 gen_name3))))
    (recv (enc nonce10 (ltk gen_name18 gen_name6)))
    (recv nonce3)
    (send (cat (ltk gen_name2 gen_name3) (cat gen_name8 nonce3)))
    (send (enc nonce3 (ltk gen_name24 gen_name25)))
    (send nonce10)))
(defrole role3
  (vars (gen_name28 name)(gen_name18 name)(gen_name31 name)(nonce1 text))
  (trace
    (send gen_name28)
    (recv (cat (pubk gen_name18) nonce1))
    (send (enc (pubk gen_name31) (pubk gen_name31)))))
(defrole role4
  (vars (gen_name13 name)(gen_name33 name)(gen_name8 name)(name1 name)(gen_name18 name)
          (name2 name)(gen_name24 name)(gen_name25 name)(name4 name)(gen_name15 name)
          (gen_name14 name)(gen_name34 name)(nonce13 text)(nonce3 text)(nonce4 text))
    (trace
      (recv nonce13)
      (send (ltk gen_name13 gen_name33))
      (recv (cat gen_name8 nonce3))
      (send nonce13)
      (send (cat name1 nonce4))
      (send gen_name18)
      (recv name2)
      (send (enc nonce3 (ltk gen_name24 gen_name25)))
      (recv (ltk name4 gen_name15))
      (send (cat (ltk gen_name14 name4) (pubk gen_name34))))))
```

B.4 Realistic Sample Protocol

```
(defprotocol protocol1 basic
  (defrole role0
    (vars (name0 name)(name1 name)(nonce1 text)(nonce2 text)(nonce3 text))
    (trace
      (send (enc (cat (ltk name0 name1) (cat nonce1 (cat name0 (cat nonce2 name0))))
            (ltk name0 name1)))
      (send (cat (ltk name0 name1) (cat name1 (enc (cat nonce1 (cat nonce2 (ltk name0
            name1))) (pubk name0)))))
      (send (enc (cat name1 (cat (ltk name0 name1) (cat nonce2 name1))) (pubk name0)))
      (send (cat (ltk name0 name1) (cat nonce1 (enc (cat name1 (cat nonce1 (ltk name0
            name1))) (pubk name1)))))
      (send (cat name0 (cat (ltk name0 name1) (cat nonce3 (enc name1 (ltk name0 name1))))))
      (send (cat nonce3 (cat (ltk name0 name1) (cat nonce3 (enc (cat name1 name1) (ltk name0
            name1))))))
      (send (cat name0 (enc (cat (ltk name0 name1) (cat nonce1 name1)) (ltk name0 name1))))
      (send (cat name1 (enc (cat (ltk name0 name1) (cat nonce3 (cat name0 (enc (enc name1
            (ltk name0 name1)) (ltk name0 name1))))) (ltk name0 name1))))))
  (defrole role1
    (vars (name0 name)(name1 name)(nonce1 text)(nonce2 text)(nonce3 text))
    (trace
      (send (enc (cat (ltk name0 name1) (cat nonce1 (cat name0 (cat nonce2 name0)))) (ltk
            name0 name1)))
      (send (cat (ltk name0 name1) (cat name1 (enc (cat nonce1 (cat nonce2 (ltk name0 name1)))
            (pubk name0)))))
      (send (enc (cat name1 (cat (ltk name0 name1) (cat nonce2 name1))) (pubk name0)))
      (send (cat (ltk name0 name1) (cat nonce1 (enc (cat name1 (cat nonce1 (ltk name0 name1)))
            (pubk name1)))))
      (send (cat name0 (cat (ltk name0 name1) (cat nonce3 (enc name1 (ltk name0 name1))))))
      (send (cat nonce3 (cat (ltk name0 name1) (cat nonce3 (enc (cat name1 name1) (ltk name0
            name1))))))
      (send (cat name0 (enc (cat (ltk name0 name1) (cat nonce1 name1)) (ltk name0 name1))))
      (send (cat name1 (enc (cat (ltk name0 name1) (cat nonce3 (cat name0 (enc (enc name1
            (ltk name0 name1)) (ltk name0 name1))))) (ltk name0 name1)))))))
```

APPENDIX C

Appendix C–Source Code

C.1 Generator.java

```
import java.util.Random;

/**
 * Top-level Generator class.  Invocation of this class creates an empty {@link Protocol},
 * then invokes the  appropriate {@link Protocol} method to populate the  protocol
 * according to the distribution specified on the command line. Expected usage:
 * {@code java Generator random_type random_seed}
 *
 * @author Stephanie Skaff
 *
 * @param random_type  Generation distribution; valid types are {@code naive}, {@code
 * bounded_naive}, {@code  * intended_naive} and {@code gaussian}.
 * @param random_seed  Number ({@code int}) used to seed
 * the Java random number generator.
 *
 *  Necessary modifications for additional realistic
 *  distributions:
 *      1) Add the name of the new distribution to the enumerated type{@code Distrib}
 *      2) Alter the error message for incorrect command line arguments to include the
 *          new distribution
 *      3) Add a case for the new distribution to the switch statement
 *      4) Alter the javadoc for random_type in this comment to include the new
 *          distribution
 *  Additional instructions for introduction of a new distribution may be found in the
 * {@link Protocol} and {@link Gaussian} classes.
 */
public class Generator {
    public static Random randInt;
    public enum Distrib {
        NAIVE, BOUNDED_NAIVE, INTENDED_NAIVE, GAUSSIAN
    }
    public static Distrib randType;

    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("Usage: Generator random_type random_seed");
            System.exit(1);  // exit generator
        }
        else {
            randType = Generator.Distrib.valueOf(args[0].toUpperCase());
            if (!(randType instanceof Distrib)) {System.err.print("Invalid random_type.  "
                                    + "Valid random_types are naive, "
                                    + "bounded_naive, intended_naive "
                                    + "and gaussian");
                System.exit(1);  // exit generator
            }
            else {
                randInt = new Random(Integer.parseInt(args[1]));
                Protocol protocol1 = new Protocol("protocol1");
                switch (randType) {
                case NAIVE:
                    protocol1.naive(); break;
                case BOUNDED_NAIVE:
                    protocol1.boundedNaive(); break;
                case INTENDED_NAIVE:
                    protocol1.intendedNaive(); break;
                case GAUSSIAN:
                    Distribution gauss = new Gaussian();
                    protocol1.realistic(gauss); break;
                }
            CPSAFile cpsa1 = new CPSAFile();
            cpsa1.protToCPSA(protocol1);
            }
        }
    }
}
```

C.2 Distribution.java

```
/**
 * Upper-level shell class for statistical distributions.
 *
 * This class contains:
 *  1)  Protocol statistics based on an analysis of the Clark-Jacob protocol library, and
 *      access functions for those statistics.  These should not be altered when creating
 *      a new distribution; alter these only if you wish to alter the underlying statistics
 *      of realistic protocol generation.
 *  2)  An abstract function which returns a random value generated according to the desired
 *      statistical distribution. This must be implemented by each extending class.
 *
 * No changes to this class are necessary to create a new realistic random distribution.
 *
 * @author Stephanie Skaff
 * @see {@link Gaussian}
 */

public abstract class Distribution {
    // roles/strands per protocol
    private static final double ROLE_MEAN = 2.688;
    private static final double ROLE_DEV = 0.535;
    // messages per protocol
    private static final double MSG_PROT_MEAN = 4.094;
    private static final double MSG_PROT_DEV = 1.838;
    // messages sent per role/strand
    private static final double MSGS_SENT_MEAN = 1.523;
    private static final double MSGS_SENT_DEV = 0.836;
    // unique names per protocol
    private static final double NAMES_UNIQ_PROT_MEAN = 2.156;
    private static final double NAMES_UNIQ_PROT_DEV = 0.369;
    // unique nonces per protocol
    private static final double NONCES_UNIQ_PROT_MEAN = 2.469;
    private static final double NONCES_UNIQ_PROT_DEV = 1.319;
    // unique keys per protocol
    private static final double KEYS_UNIQ_PROT_MEAN = 0.813;
    private static final double KEYS_UNIQ_PROT_DEV = 0.738;
    // unique atoms per protocol
    private static final double ATOMS_UNIQ_PROT_MEAN = 6.156;
    private static final double ATOMS_UNIQ_PROT_DEV = 2.398;
    // depth of message nesting
    private static final double MSG_DEPTH_MEAN = 3.718;
    private static final double MSG_DEPTH_DEV = 1.962;
    // unique names per message
    private static final double NAMES_UNIQ_MSG_MEAN = 1.115;
    private static final double NAMES_UNIQ_MSG_DEV = 0.874;
    // unique nonces per message
    private static final double NONCES_UNIQ_MSG_MEAN = 1.305;
    private static final double NONCES_UNIQ_MSG_DEV = 0.722;
    // unique keys per message
    private static final double KEYS_UNIQ_MSG_MEAN = 0.254;
    private static final double KEYS_UNIQ_MSG_DEV = 0.471;
    // unique atoms per message
    private static final double ATOMS_UNIQ_MSG_MEAN = 2.87;
    private static final double ATOMS_UNIQ_MSG_DEV = 1.541;
     // total atoms per message
    private static final double ATOMS_TOTAL_MSG_MEAN = 3.313;
    private static final double ATOMS_TOTAL_MSG_DEV = 2.253;
    // encryptions per message
    private static final double ENC_MSG_MEAN = 0.817;
    private static final double ENC_MSG_DEV = 0.688;
```

```
/**
 * This function must be implemented for each new distribution.
 * @return a random value generated according to the desired distribution
 */
public abstract int getRandomInt(double mean, double dev);


/* *************************************************
 *
 * Retrieval classes for distribution values.
 *
 * Do not alter these when making a new distro!
 *
 **************************************************/

/**
 * Returns the average number of roles per protocol
 */
public double getRoleMean() {
    return ROLE_MEAN;
}

/**
 * Returns the standard deviation of the number of roles per protocol
 */
public double getRoleDev() {
    return ROLE_DEV;
}

/**
 * Returns the average number of messages per protocol
 */
public double getMsgProtMean() {
    return MSG_PROT_MEAN;
}

/**
 * Returns the standard deviation of the number of messages per protocol
 */
public double getMsgProtDev() {
    return MSG_PROT_DEV;
}

/**
 * Returns the average number of messages sent per role. (This is equivalent to the
 * {@link Node}s whose direction is "-" or SENT.)
 */
public double getMsgsSentMean() {
    return MSGS_SENT_MEAN;
}

/**
 * Returns the standard deviation of the number of messages sent per role.  (This is
 * equivalent to the {@link Node}s whose direction is "-" or SENT.)
 */
public double getMsgsSentDev() {
    return MSGS_SENT_DEV;
}

/**
 * Returns the average number of unique {@link Name}s per protocol
 */
public double getNamesUniqProtMean() {
    return NAMES_UNIQ_PROT_MEAN;
}

/**
 * Returns the standard deviation of the number of unique {@link Name}s per protocol
 */
public double getNamesUniqProtDev() {
    return NAMES_UNIQ_PROT_DEV;
}

/**
 * Returns the average number of unique nonces per protocol
 */
public double getNoncesUniqProtMean() {
    return NONCES_UNIQ_PROT_MEAN;
}

/**
 * Returns the standard deviation of the number of
 * unique nonces per protocol
 */
public double getNoncesUniqProtDev() {
    return NONCES_UNIQ_PROT_DEV;
}

/**
 * Returns the average number of unique keys transmitted
 * per protocol
 */
public double getKeysUniqProtMean() {
    return KEYS_UNIQ_PROT_MEAN;
}

/**
 * Returns the standard deviation of the number of unique
 * keys transmitted per protocol
 */
public double getKeysUniqProtDev() {
    return KEYS_UNIQ_PROT_DEV;
}

/**
 * Returns the average number of unique atoms per protocol
```

```
     */
    public double getAtomsUniqProtMean() {
        return ATOMS_UNIQ_PROT_MEAN;
    }

    /**
     * Returns the standard deviation of the number of
     * unique atoms per protocol
     */
    public double getAtomsUniqProtDev() {
        return ATOMS_UNIQ_PROT_DEV;
    }

    /**
     * Returns the average nesting depth of message
     * components
     */
    public double getMsgDepthMean() {
        return MSG_DEPTH_MEAN;
    }

    /**
     * Returns the standard deviation of the nesting depth
     * of message components
     */
    public double getMsgDepthDev() {
        return MSG_DEPTH_DEV;
    }

    /**
     * Returns the average number of unique names per message
     */
    public double getNamesUniqMsgMean() {
        return NAMES_UNIQ_MSG_MEAN;
    }

    /**
     * Returns the standard deviation of the number of
     * unique names per message
     */
    public double getNamesUniqMsgDev() {
        return NAMES_UNIQ_MSG_DEV;
    }

    /**
     * Returns the average number of unique nonces per
     * message
     */
    public double getNoncesUniqMsgMean() {
        return NONCES_UNIQ_MSG_MEAN;
    }

    /**
```

```
 * Returns the standard deviation of the number of
 * unique nonces per message
 */
public double getNoncesUniqMsgDev() {
    return NONCES_UNIQ_MSG_DEV;
}

/**
 * Returns the average number of unique keys transmitted per message
 */
public double getKeysUniqMsgMean() {
    return KEYS_UNIQ_MSG_MEAN;
}

/**
 * Returns the standard deviation of the number of unique keys transmitted per
 * message
 */
public double getKeysUniqMsgDev() {
    return KEYS_UNIQ_MSG_DEV;
}

/**
 * Returns the average number of unique atoms per message
 */
public double getAtomsUniqMsgMean() {
    return ATOMS_UNIQ_MSG_MEAN;
}

/**
 * Returns the standard deviation of the number of unique atoms per message
 */
public double getAtomsUniqMsgDev() {
    return ATOMS_UNIQ_MSG_DEV;
}

/**
 * Returns the average total number of atoms per message
 */
public double getAtomsTotalMsgMean() {
    return ATOMS_TOTAL_MSG_MEAN;
}

/**
 * Returns the standard deviation of the total number of atoms per message
 */
public double getAtomsTotalMsgDev() {
    return ATOMS_TOTAL_MSG_DEV;
}

/**
 * Returns the average number of encryptions per message
 */
```

```
    public double getEncMsgMean() {
        return ENC_MSG_MEAN;
    }

    /**
     * Returns the standard deviation of the number of encryptions per message
     */
    public double getEncMsgDev() {
        return ENC_MSG_DEV;
    }
}
```

C.3 Gaussian.java

```
/**
 * The Gaussian class implements the single abstract function of the {@link Distribution}
 * class, {@code getRandomInt}.  To create a new realistic distribution, the user should
 * create a new class using this class as a template.  The only necessary change to the
 * class (besides the name) is to replace the use of {@link Generator.randInt.nextGaussian()}
 * with the desired random function.
 *
 * In addition to creating a class for the new distribution, please see the {@link Generator}
 * class for further instructions.
 *
 * @author Stephanie Skaff
 * @see {@link Distribution}
 */
public class Gaussian extends Distribution {

    /**
     * Returns the next random value for this distribution.
     * @return a non-zero random z-score
     */
    public int getRandomInt(double mean, double dev) {
        int returnVal = -1;
        /*
         * replace the following line when implementing a new distribution!
         */
        double nextRand = Generator.randInt.nextGaussian();
        while (returnVal <= 0) {
            returnVal = (int) Math.round((nextRand * dev) + mean);
        }
        return returnVal;
    }
}
```

C.4 CPSAFile.java

```java
import java.io.FileDescriptor;
import java.io.FileOutputStream;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Iterator;

import java.io.FileDescriptor;
import java.io.FileOutputStream;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Iterator;

/**
 * Outputs a generated protocol in CPSA format
 *
 * @author stephanie
 *
 */
public class CPSAFile {
    private FileOutputStream outFileStream;
    private PrintWriter outStream;

    /**
     * Opens a {@link PrintWriter} for outputting the CPSA-format protocol as a text file.
     */
    public CPSAFile() {
        outFileStream = new FileOutputStream(FileDescriptor.out);
        outStream = new PrintWriter(outFileStream);
    }

    /**
     * Outputs a CPSA preskeleton based on the {@link Protocol} provided. This function is
     * still in development.
     * @param prot {@link Protocol} to use for preskeleton.  This currently uses only two
     * {@link Strand}s in the preskeleton.
     */
    public void makePreSkeleton(Protocol prot) {
        Atom tempAtom;
        outStream.println("(defpreskeleton " + prot.getName());

        // select skeleton and test strands
        int skelInt = Generator.randInt.nextInt(prot.getLength());
        Strand skelStrand = prot.getStrand(skelInt);
        int other = Generator.randInt.nextInt(prot.getLength());
        while (skelInt == other) {
            other = Generator.randInt.nextInt(prot.getLength());
        }
        Strand otherStrand = prot.getStrand(other);

        Iterator<Atom> eachAtom = skelStrand.iterAtoms();
        ArrayList<Nonce> nonces = skelStrand.getNewNonces();
        outStream.print("  (vars ");
            //("  + skelStrand.getRoleName() +  " name) ");

        eachAtom = skelStrand.iterSentAtoms();
        String typeString = "";
        String valString = "";
        /*fix: if no nonces in list, nextInt(0) returns any
         * random int (not bounded 0-size)
         */
        Nonce listenForThis = nonces.get(Generator.randInt.nextInt(nonces.size()));
        while (eachAtom.hasNext()) {
            tempAtom = eachAtom.next();
            if (tempAtom instanceof Name) {
                Name tempName = (Name) tempAtom;
                typeString = "name";
                valString = tempName.getName();

            }
            else if (tempAtom instanceof Nonce) {
                typeString = "text";
                Nonce tempNonce = (Nonce) tempAtom;
                valString = tempNonce.getName();
            }
            outStream.print("(" + valString + " " + typeString + ")");
        }

        outStream.println(")");
        //needs other strand action!
        outStream.println("  (defstrand " + otherStrand.getRoleID() + " "
                          + otherStrand.getLength() + " ("
                          + otherStrand.getRoleName() + " " + skelStrand.getRoleName()
                          + ") " + "(" + listenForThis.getName() + " "
                          + listenForThis.getName() + ")))");
        /*outStream.println("  (deflistener " + listenForThis.getName() +")");
        //uniq-orig
        //outStream.println("  (uniq-orig " + listenForThis.getName() + ")");

        //outStream.println("  (non-orig (privk " + skelStrand.getRoleName() + ")))"); */
    }


    /**
     * Converts a {@link Protocol} into CPSA format.
     *
     * @param prot {@link Protocol} to be converted
     */
    public void protToCPSA(Protocol prot) {
        Strand tempStrand;
        outStream.print("(defprotocol " + prot.getName() + " basic");
        Iterator<Strand> eachStrand = prot.iterStrands();
        while (eachStrand.hasNext()) {
            tempStrand = eachStrand.next();
            this.strandToCPSA(tempStrand);
        }
        outStream.println(")\n");
        //this.makePreSkeleton(prot);
        outStream.close();
    }

    /**
     * Converts a {@link Strand} into CPSA format.
     * @param strnd {@link Strand} to be converted
     */
    public void strandToCPSA(Strand strnd) {
        Node tempNode;
        Atom tempAtom;
        String typeString = "";
        String valString = "";
        outStream.println("\n  (defrole " + strnd.getRoleID());

        outStream.print("    (vars ");
        Iterator<Atom> eachAtom = strnd.iterAtoms();
        while (eachAtom.hasNext()) {
            tempAtom = eachAtom.next();
            if (tempAtom instanceof Name) {
                typeString = "name";
                Name tempName = (Name) tempAtom;
                valString = tempName.getName();
                outStream.print("(" + valString + " " + typeString + ")");
            }
            else if (tempAtom instanceof Nonce) {
                typeString = "text";
                Nonce tempNonce = (Nonce) tempAtom;
                valString = tempNonce.getName();
                outStream.print("(" + valString + " " + typeString + ")");
            }
            else {  // tempAtom is a Key
                // need to add the keys! or dump for CPSA?
                /*typeString = "text";
                Nonce tempNonce = (Nonce) tempAtom;
                valString = tempNonce.getName();
                outStream.print("(" + valString + " " + typeString + ")"); */
            }
        }

        outStream.print(")\n    (trace");
        Iterator<Node> eachNode = strnd.iterNodes();
        while (eachNode.hasNext()) {
            tempNode = eachNode.next();
            this.nodeToCPSA(tempNode);
        }
        outStream.print(")");
        // uniq-orig goes here!!!! ****
        outStream.print(")");
```

```
}

/**
 * Converts a {@link Node} to CPSA form
 * @param node {@link Node} to be converted
 */
public void nodeToCPSA(Node node) {
    outStream.print("\n      (");
    if (node.getDirection().equals("+"))
        outStream.print("send ");
    else
        outStream.print("recv ");
    this.msgToCPSA(node.getBody());
    outStream.print(")");
}

/**
 * Converts a {@link Message} to CPSA format
 * @param msg {@link Message} to be converted
 */
public void msgToCPSA(Message msg) {
    if (msg instanceof Encryption) {
        this.encToCPSA((Encryption) msg);
    }
    else if (msg instanceof Cons) {
        this.consToCPSA((Cons) msg);
    }
    else if (msg instanceof Atom) {
        this.atomToCPSA((Atom) msg);
    }
}

/**
 * Converts an {@link Encryption} to CPSA format
 * @param enc {@link Encryption} to be converted
 */
public void encToCPSA(Encryption enc) {
    outStream.print("(enc ");
    this.msgToCPSA(enc.getContents());
    outStream.print(" ");
    this.keyToCPSA(enc.getKey());
    outStream.print(")");
}

/**
 * Converts an {@link Cons} to CPSA format
 * @param cons {@link Cons} to be converted
 */
public void consToCPSA(Cons cons) {
    outStream.print("(cat " );
    this.msgToCPSA(cons.getFirst());
    outStream.print(" ");
    this.msgToCPSA(cons.getSecond());
    outStream.print(")");
}

/**
 * Converts an {@link Atom} to CPSA format
 * @param atom {@link Atom} to be converted
 */
public void atomToCPSA(Atom atom) {
    if (atom instanceof Name) {
    this.nameToCPSA((Name) atom);
    }
    else if (atom instanceof Nonce) {
        this.nonceToCPSA((Nonce) atom);
    }
    else if (atom instanceof Key) {
        this.keyToCPSA((Key) atom);
    }
}

/**
 * Converts a {@link Name} to CPSA format
 * @param name {@link Name} to be converted
 */
public void nameToCPSA(Name name) {
    outStream.print(name.getName());
}

/**
 * Converts a {@link Nonce} to CPSA format
 * @param nonce {@link Nonce} to be converted
 */
public void nonceToCPSA(Nonce nonce) {
    outStream.print(nonce.getName());
}

/**
 * Converts a {@link Key} to CPSA format
 * @param key {@link Key} to be converted
 */
public void keyToCPSA(Key key) {
    /*if (key instanceof Encryptor) {
        this.encryptorToCPSA((Encryptor) key);
    }
    else */
    if (key instanceof PublicKey) {
        this.pubKeyToCPSA((PublicKey) key);
    }
    else if (key instanceof PrivateKey) {
        this.privKeyToCPSA((PrivateKey) key);
    }
    else if (key instanceof LTPairKey) {
        this.ltPairKeyToCPSA((LTPairKey) key);
    }
```

```
        else if (key instanceof SessionKey) {
            this.sessionKeyToCPSA((SessionKey) key);
        }
    }

    /**
     * Converts an {@link Encryptor} to CPSA format
     * @param key {@link Encryptor} to be converted
     */
    public void encryptorToCPSA(Encryptor key) {
        if (key instanceof PublicKey) {
            this.pubKeyToCPSA((PublicKey) key);
        }
        else if (key instanceof LTPairKey) {
            this.ltPairKeyToCPSA((LTPairKey) key);
        }
        else if (key instanceof SessionKey) {
            this.sessionKeyToCPSA((SessionKey) key);
        }   }

    /**
     * Converts a {@link PublicKey} to CPSA format
     * @param pubk {@link PublicKey} to be converted
     */
    public void pubKeyToCPSA(PublicKey pubk) {
        outStream.print("(pubk " + pubk.getKeyholder() + ")");
    }

    /**
     * Converts a {@link PrivateKey} to CPSA format
     * @param privk {@link PrivateKey} to be converted
     */
    public void privKeyToCPSA(PrivateKey privk) {
        outStream.print("(privk " + privk.getKeyholder() + ")");
    }

    /**
     * Converts a {@link LTPairKey} to CPSA format
     * @param ltpk {@link LTPairKey} to be converted
     */
    public void ltPairKeyToCPSA(LTPairKey ltpk) {
        outStream.print("(ltk " + ltpk.getKeyholder1().toString() + " "
                                + ltpk.getKeyholder2().toString() + ")");
    }


    //CPSA does not utilize session keys as of 1.0; output
    //as ltp keys
    public void sessionKeyToCPSA(SessionKey sessk) {
        outStream.print("(ltk " + sessk.getKeyholder1().toString() + " "
                                + sessk.getKeyholder2().toString() + ")");
    }
}
```

C.5 Protocol.java

```
/**
 *   Second level of the Generator; top level for actual protocol generation.
 *
 *   @author Stephanie Skaff
 */

import java.util.*;

public class Protocol {
    private String protocolName;
    private ArrayList<Strand> strands;
    public ArrayList<Atom> allAtoms;

    /**
     * Creates an empty Protocol
     *
     * @param protName unique name of new Protocol
     */
    public Protocol(String protName) {
        protocolName = protName;
        strands = new ArrayList<Strand>();
        allAtoms = new ArrayList<Atom>();
    }

    /**
     * Creates a Protocol with {@code numRoles} roles. Currently not used in the
     * Generator class.
     *
     * @param protName unique name of new Protocol
     * @param numRoles number of roles in the new Protocol
     */
    public Protocol(String protName, int numRoles) {
        Strand newStrand;
        allAtoms = new ArrayList<Atom>();
        protocolName = protName;

        strands = new ArrayList<Strand>();
        for (int count = 0; count < numRoles; count++){
            //make each strand here
            newStrand = new Strand(count);
            strands.add(newStrand);
        }
    }


    /* ****************************************************
     *
     *  Distribution creation functions -
     *      called after creating an empty Protocol
     *
     ****************************************************/
```

```
public void realistic(Distribution distro) {
    Strand strand1, strand2;
    int count, nodeCount;


    int numRoles, numNodes, numNames;
    int numNonces, numKeys;
    Node tempNode, tempNode2;
    Nonce tempNonce;
    Name tempName, tempName2;
    Key tempKey;

    /*choose protocol-level parameters and source list
    for message components  */
    do {
        numRoles = distro.getRandomInt(distro.getRoleMean(), distro.getRoleDev());
    }
    while (numRoles < 2);

    for (count = 0; count < numRoles; count++){
            strand1 = new Strand(count);
            tempKey = new PublicKey(strand1.getRoleName());
            strands.add(strand1);
    }

    numNames = distro.getRandomInt(distro.getNamesUniqProtMean(),
                                distro.getNamesUniqProtDev());
    if (numRoles > numNames) {
        // we need more names than we have roles
        for (count = numNames; count < numRoles; count++) {
            tempName = new Name(Name.getNewNameString());
        }
    }
    //make all nonces
    numNonces = distro.getRandomInt(distro.getNoncesUniqProtMean(),
                                 distro.getNoncesUniqProtDev());
    for (count = 0; count < numNonces; count++) {
        tempNonce = new Nonce();
    }

    // make all transmitted keys
    numKeys = distro.getRandomInt(distro.getKeysUniqProtMean(),
                                 distro.getKeysUniqProtDev());
    for (count = 0; count < numKeys;count++) {
        //make transmitted keys here - will always be symKeys
        tempName = Name.names.get(Generator.randInt.nextInt(Name.names.size()));
        do {
            tempName2 = Name.names.get(Generator.randInt.nextInt(Name.names.size()));
        }
        while (tempName2.equals(tempName));
        tempKey = new SessionKey(tempName, tempName2);
    }
    // load strands
    numNodes = distro.getRandomInt(distro.getMsgProtMean(), distro.getMsgProtDev());

    int str1, str2;
    for (nodeCount = 0; nodeCount < numNodes; nodeCount++) {
        str1 = Generator.randInt.nextInt(strands.size());
        strand1 = strands.get(str1);
        str2 = Generator.randInt.nextInt(strands.size());
        while (str1 == str2) {
            str2 = Generator.randInt.nextInt(strands.size());
        }
        strand2= strands.get(str2);
        tempNode = new Node();
        tempNode.realistic(distro);
        tempNode.setDirection("+");
        strand1.setLast(tempNode);
        tempNode2 = new Node();
        tempNode2.setBody(tempNode.getBody());
        tempNode2.setDirection("-");
        strand2.setLast(tempNode);
    }
    // give a node to any strand which didn't get one
    //via random process (for benefit of protocol
    //verifiers)
    for (count = 0; count < strands.size(); count ++){
        strand1 = strands.get(count);
        if (strand1.getNodeCount() == 0) {
            do {
                str2 = Generator.randInt.nextInt(strands.size());
            }
            while (str2 == count);
            strand2 = strands.get(str2);
            double dirRand = Math.random();
            tempNode = new Node();
            tempNode.realistic(distro);
            if (dirRand < .5)
                tempNode.setDirection("+");
            else
                tempNode.setDirection("-");
            strand1.setLast(tempNode);
            tempNode2 = new Node();
            tempNode2.setBody(tempNode.getBody());
            if (dirRand < .5)
                tempNode.setDirection("-");
            else
                tempNode.setDirection("+");
            strand2.setLast(tempNode);
        }
    }
}
```

```
/**
 * Fills this {@link Protocol} with an intended run composed of uniformly distributed
 * contents
 */
public void intendedNaive() {
    Strand strand1, strand2;
    int numNodes, count, numRoles = 1;
    Message newMessage;
    Node tempNode;
    //choose number of strands >=2
    do {
        numRoles = Generator.randInt.nextInt(20);
    }
    while (numRoles < 2);
    // make strands, but do not fill them
    for (count = 0; count < numRoles; count++){
            strand1 = new Strand(count);
            strands.add(strand1);
    }

    // load strands
    numNodes = 0;
    do {
        numNodes = Generator.randInt.nextInt(20);
    }
    while (numNodes < 1);

    int str1, str2;
    int[] strCount =  new int[strands.size()];
    for (count = 0; count < strands.size(); count++) {
        strCount[count] = count;
    }
    for (count = 0; count < numNodes; count++) {
        newMessage = Message.naive();
        str1 = Generator.randInt.nextInt(strands.size());
        strand1 = strands.get(str1);
        str2 = Generator.randInt.nextInt(strands.size());
        while (str1 == str2) {
            str2 = Generator.randInt.nextInt(strands.size());
        }
        strand2= strands.get(str2);
        tempNode = new Node();
        tempNode.setBody(newMessage);
        tempNode.setDirection("+");
        strand1.setLast(tempNode);
        tempNode = new Node();
        tempNode.setBody(newMessage);
        tempNode.setDirection("-");
        strand2.setLast(tempNode);
    }
    // give a node to any strand which didn't get one
    //via random process (for benefit of protocol
    //verifiers)
    for (count = 0; count < strands.size(); count ++){
        strand1 = strands.get(count);
        if (strand1.getNodeCount() == 0) {
            newMessage = Message.naive();
            do {
                str2 = Generator.randInt.nextInt(strands.size());
            }
            while (str2 == count);
            strand2 = strands.get(str2);
            double dirRand = Math.random();
            tempNode = new Node();
            tempNode.setBody(newMessage);
            if (dirRand < .5)
                tempNode.setDirection("+");
            else
                tempNode.setDirection("-");
            strand1.setLast(tempNode);
            tempNode = new Node();
            tempNode.setBody(newMessage);
            if (dirRand < .5)
                tempNode.setDirection("-");
            else
                tempNode.setDirection("+");
            strand2.setLast(tempNode);
        }
    }
}


/**
 * Fills a {@link Protocol} according to a bounded
 * naive/uniform distribution of contents.
 */
public void boundedNaive() {
    Strand newStrand;
    int numRoles = 0;
    do {
        numRoles = Generator.randInt.nextInt(20);
    }
    while (numRoles <= 0);

    // make and fill strands
    for (int count = 0; count < numRoles; count++){
            newStrand = new Strand(count);
            newStrand.boundedNaive();
            strands.add(newStrand);
    }
}

/**
 * Fills a {@link Protocol} according to an unbounded
 * naive/uniform distribution of contents.
 */
```

```
public void naive() {
    Strand newStrand;
    int numRoles = 0;

    //choose number of strands
    do {
        // arbitrary top of 100 to enforce positive int
        numRoles = Generator.randInt.nextInt(100);
    }
    while (numRoles == 0);

    // make strands
    for (int count =0; count < numRoles; count++) {
        newStrand = new Strand(count);
        strands.add(newStrand);
    }
    for (int count = 0; count < numRoles; count++){
        newStrand = strands.get(count);
        newStrand.naive();
    }
}


/* **************************************************
 *
 *  Protocol handling functions
 *
 ***************************************************/


/**
 * Returns String containing name of protocol
 *
 * @return name of protocol
 */

public String getName() {
    return protocolName;
}

/**
 * Returns an Iterator for the Strands of the Protocol
 *
 *  @return Strand Iterator
 */

public Iterator<Strand> iterStrands() {
    return strands.iterator();
}

/**
 * Returns the Strand at index {@code index}
 *
```

```
     *  @param index integer representing the requested Strand
     *  @return Strand Iterator
     */

    public Strand getStrand(int index) {
        return strands.get(index);
    }

    /**
     * Sets the Strand
     *
     *  @param index integer at which to place {@code newStrand}
     *  @param newStrand Strand to be placed at index {@code index}
     */

    public void setStrand(int index, Strand newStrand) {
        strands.add(index, newStrand);
    }

    /**
     * Returns number of {@link Strand}s in the Protocol
     * @return number of Strands
     */
    public int getLength() {
        return strands.size();
    }

    /**
     *  Converts protocol into a viewable format
     */
    public void output() {
        System.out.println(protocolName + "\t");
        for (int outCount = 0; outCount < this.getLength(); outCount++) {
            strands.get(outCount).output();
        }
    }
}
```

C.6 Strand.java

```
import java.util.*;

/**
 * A strand contains a series of {@link Node} objects which represent messages sent or
 * received by this Strand.
 *
 * @author Stephanie
 */
public class Strand {
    private String roleID;
private Name roleName;
private LinkedList<Node> nodes;
//private LinkedList<Atom> atoms;

/**
 * Constructor taking a name and returning a Strand.
 * @param aName
 * @return the new Strand
 */
public Strand (String aName) {
    roleName = new Name(aName);
    nodes = new LinkedList<Node>();
    //atoms.add(roleName);
}

/**
 * Creates a new {@link Strand} numbered {@code roleNum}
 * @param roleNum integer to use as part of the {@code roleName}
 */
public Strand (int roleNum) {
    roleID = "role" + roleNum;
    roleName = new Name("name"+ roleNum);
    nodes = new LinkedList<Node>();
}

/**
 * Creates a new {@link Strand} numbered {@code roleNum}, containing {@code length}
 * {@link Nodes}
 * @param roleNum integer to use as part of the {@code roleName}
 * @param length number of nodes in this {@link Strand}
 */
public Strand (int roleNum, int length) {
    roleID = "role" + roleNum;
    roleName = new Name("name"+ roleNum);
    nodes = new LinkedList<Node>();
    Node newNode;
    for (int i = 0; i < length; i++) {
        // make new nodes & add to list
        newNode = new Node();
        nodes.add(newNode);
    }
}

/**
 * Populates a {@link Strand} according to an unbounded naive distribution
 */
public void naive() {
    Node newNode;
    int numNodes = 0;
    //choose number of nodes on this strand
    do {
        numNodes = Generator.randInt.nextInt(100);
    }
    while (numNodes <= 0);

    // make nodes for this strand
    for (int count = 0; count < numNodes; count++){
            newNode = new Node();
            newNode.naive();
            nodes.add(newNode);
    }
}

/**
 * Populates a {@link Strand} according to a bounded-naive distribution.
 */
public void boundedNaive() {
    Node newNode;
    int numNodes = 0;
    do {
        numNodes = Generator.randInt.nextInt(5);
    }
    while (numNodes <= 0);

    // make strands
    for (int count = 0; count < numNodes; count++){
        //make each strand here
            newNode = new Node();
            newNode.boundedNaive();
            nodes.add(newNode);
    }
}

/**
 * Return the roleID String of the current {@link Strand}
 * @return string containing the role id
 */
public String getRoleID(){
    return roleID;
}

/**
 * Sets the {@code roleID} to the provided string
 * @param newId String containing the new id
 */
public void setID(String newID){
    roleID = newID;
}

/**
 * Return the roleName String of the current Name
 * @return string containing the rolename
 */
public Name getRoleName(){
    return roleName;
}
```

```
/**
 * Sets the {@code roleName} to the provided string
 * @param newName String representation of the {@link Name}
 */
public void setRoleName(String newName){
    roleName = new Name(newName);
}


/**
 * Retrieves the Node from {@link nodes} at index {@code i}
 * @param i Index into array of Nodes
 * @return Node at index I
 */
public Node getNode(int i){
    return nodes.get(i);
}

/**
 * Returns the number of {@link Node}s associated with this {@link Strand}
 * @return number of nodes
 */
public int getNodeCount() {
    return nodes.size();
}

/**
 * Adds {@code newNode} to {@code nodes} at position {@code i}.
 * @param i Index at which to insert this {@link Node} into {@code nodes}
 * @param newNode {@link Node} to insert
 */
public void setNode(int i, Node newNode) {
    nodes.add(i, newNode);
}

/**
 * Adds {@code newNode} to {@code nodes} at the end of the list.
 * @param newNode {@link Node} to insert
 */
public void setLast(Node newNode) {
    nodes.addLast(newNode);
}

/**
 * Returns an iterator through this {@link Strand}'s {@link Node}s.
 * @return iterator for this {@link Strand}'s {@link Node}s.
 */
public Iterator<Node> iterNodes() {
    return nodes.iterator();
}

/**
```

```
 * Returns a list of all {@link Atom}s used in the {@link Message}s of this {@link
 * Strand}
 * @return list of Atoms appearing inside this {@link Strand}
 */
public Iterator<Atom> iterAtoms() {
    ArrayList<Atom> varList = new ArrayList<Atom>();
    ArrayList<Name> nameList = new ArrayList<Name>();
    ArrayList<Nonce> nonceList = new ArrayList<Nonce>();
    ArrayList<Key> keyList = new ArrayList<Key>();
    Iterator<Atom> iterVars;
    Iterator<Name> iterNames;
    Iterator<Nonce> iterNonces;
    Iterator<Key> iterKeys;
    Iterator<Node> iterNode = nodes.iterator();
    Node tempNode;
    Atom tempAtom;
    Name tempName;
    Nonce tempNonce;
    Key tempKey;
    boolean addIt = true;

    while (iterNode.hasNext()){
        tempNode = iterNode.next();
        iterVars = tempNode.getNodeVars().iterator();
        while  (iterVars.hasNext()) {
            addIt= true;
            tempAtom = iterVars.next();
            if (tempAtom instanceof Name) {
                iterNames = nameList.iterator();
                if (!(iterNames.hasNext())) {
                    nameList.add((Name) tempAtom);
                }
                else {
                    while (iterNames.hasNext()) {
                        tempName = iterNames.next();
                        if (tempName.equals((Name) tempAtom)) {
                            addIt = false;
                        }
                    }
                    if (addIt) {
                        nameList.add((Name) tempAtom);
                    }
                }
            }
            else if (tempAtom instanceof Nonce) {
                iterNonces= nonceList.iterator();
                if (!(iterNonces.hasNext())) {
                    tempNonce = (Nonce) tempAtom;
                    nonceList.add((Nonce) tempAtom);
                }
                else {
                    while (iterNonces.hasNext()) {
                        tempNonce = iterNonces.next();
                    if (tempNonce.equals((Nonce) tempAtom)) {
                        addIt = false;
                    }
                }
                if (addIt) {
                    tempNonce = (Nonce) tempAtom;
                    nonceList.add((Nonce) tempAtom);
                }
            }
        }
        else {  // Atom must be a Key-
            iterKeys= keyList.iterator();
            if (!(iterKeys.hasNext())) {
                addIt = true;
                keyList.add((Key) tempAtom);
            }
            else {
                while (iterKeys.hasNext()) {
                    tempKey = iterKeys.next();
                    if (tempKey.equals(tempAtom)) {
                        addIt = false;
                    }
                }
            }
            if (addIt) {
                tempKey = (Key) tempAtom;
                keyList.add(tempKey);
                if (tempKey instanceof SymKey) {
                    // 2 keyholders
                    SymKey tempSym = (SymKey) tempKey;
                    tempName = tempSym.getKeyholder1();
                    iterNames= nameList.iterator();
                    if (Name.checkUniqueName(tempName, iterNames))
                        nameList.add(tempName);
                    tempName = tempSym.getKeyholder2();
                    iterNames= nameList.iterator();
                    if (Name.checkUniqueName(tempName, iterNames))
                        nameList.add(tempName);
                }
                else {
                    AsymKey tempAsym = (AsymKey) tempKey;
                    tempName = tempAsym.getKeyholder();
                    iterNames= nameList.iterator();
                    if (Name.checkUniqueName(tempName, iterNames))
                        nameList.add(tempName);
                }
            }
        }
    }
}
//now combine all vars into one list
iterNames = nameList.iterator();
while (iterNames.hasNext()) {
```

```
            varList.add(iterNames.next());
        }
        iterNonces = nonceList.iterator();
        while (iterNonces.hasNext()) {
            varList.add(iterNonces.next());
        }
        iterKeys = keyList.iterator();
        while (iterKeys.hasNext()) {
            varList.add(iterKeys.next());
        }
        return varList.iterator();
    }

    public Iterator<Atom> iterSentAtoms() {
        ArrayList<Atom> varList = new ArrayList<Atom>();
        //ArrayList<Atom> tempVars;
        Iterator<Atom> iterVars;
        Iterator<Node> iterNode = nodes.iterator();
        Node tempNode;
        Atom tempAtom;

        varList.add(this.getRoleName());
        while (iterNode.hasNext()){
            tempNode = iterNode.next();
            if (tempNode.getDirection().equals("+")) {
                iterVars = tempNode.getNodeVars().iterator();
                while  (iterVars.hasNext()) {
                    tempAtom = iterVars.next();
                    if (!(varList.contains(tempAtom)))
                        if (tempAtom instanceof Name)
                        varList.add(tempAtom);
                }
            }
        }
        return varList.iterator();
    }

    public ArrayList<Nonce> getNewNonces() {
        ArrayList<Nonce> varList = new ArrayList<Nonce>();
        Iterator<Atom> iterVars;
        Iterator<Node> iterNode = nodes.iterator();
        Node tempNode;
        Atom tempAtom;

        while (iterNode.hasNext()){
            tempNode = iterNode.next();
            if (tempNode.getDirection().equals("+")) {
                iterVars = tempNode.getNodeVars().iterator();
                while  (iterVars.hasNext()) {
                    tempAtom= iterVars.next();
                    if ((!(varList.contains(tempAtom))) && (tempAtom instanceof Nonce))
                        varList.add((Nonce) tempAtom);
                }
            }
        }
        return varList;
    }

    /**
     * Returns the number of {@link Node}s in this {@link Strand}
     * @return the number of {@link Node}s in {@code node}
     */
    public int getLength() {
        return nodes.size();
    }

    /**
     * Outputs a String representation of this {@code Strand}
     */
    public void output() {
        System.out.println("strand " + roleName.getName());
        for (int nodeCount = 0; nodeCount < this.getLength(); nodeCount++) {
            System.out.print("\t");
            nodes.get(nodeCount).output();
        }
    }

    /**
     * toString - returns single String representing the entire {@link Strand}
     */
    public String toString() {
        String returnString = "strand" + roleName + "\n\t";
        for (int nodeCount = 0; nodeCount < nodes.size(); nodeCount++) {
            returnString.concat(nodes.get(nodeCount).toString() + "\n\t");
        }
        returnString.concat("\n");
        return returnString;
    }
}
```

C.7 Node.java

```
import java.util.ArrayList;
import java.util.Iterator;

/**
 * Represents a single {@link Message} and its {@code Direction} (either SEND or RECV) on a
 * {@link Strand}.
 * @author Stephanie Skaff
 */
public class Node {
    // data members
    public static enum Direction {SEND,RECV}
    private Direction sign;
```

```java
private Message body;

// constructors
public Node() {}


/**
 * Creates a {@link Node} with a specified {@code Direction} and {@link Message}.
 * @param thisway Direction of this message.  Valid values are SEND and RECV.
 * @param filler {@link Message} contained in this {@link Node}.
 */
public Node(String thisway, Message filler) {
    setDirection(thisway);
    body = filler;
}

/* ****************************************************
 *
 *  Random distribution functions at node level
 *
 *****************************************************/

/**
 * Fills a {@link Node} with a random {@link Direction} and an unbounded random
 * {@link Message}.
 */

public void naive() {
    //body = Message.naive();
    double nodeRand = Math.random();
    if (nodeRand < 0.5)
        sign = Direction.SEND;
    else
        sign = Direction.RECV;
    // random contents
    nodeRand = Math.random();
    if ((nodeRand < .5) || (Message.getCount() == 0))
        //make new Message
        body = Message.naive();
    else {
        // retrieve message from messages list
        body = Message.getMessage(Generator.randInt.nextInt(Message.getCount()));
    }
}

/**
 * Fills a {@link Node} with a random {@link Direction} and a bounded random {@link
 * Message}.
 */
public void boundedNaive() {
    body = Message.boundedNaive();
    double nodeRand = Math.random();
    if (nodeRand < 0.5)
```

```
            sign = Direction.SEND;
        else
            sign = Direction.RECV;
        // random contents
        nodeRand = Math.random();
        if ((nodeRand < .5) || (Message.getCount() == 0))
            //make new Message
            body = Message.boundedNaive();
        else {
            // retrieve message from messages list
            body = Message.getMessage(Generator.randInt.nextInt(Message.getCount()));
        }

    }


    /* ****************************************************
     * Node control functions
     *****************************************************/

    /**
     * Sets the {@link Node} direction to {@code thisway}
     * @param thisway Specifies new node direction
     */

    public void setDirection(String thisway) {
        if (thisway.equalsIgnoreCase("+")) {
            sign = Direction.SEND;
        }
        else if (thisway.equalsIgnoreCase("-"))
            sign = Direction.RECV;
    }

    /**
     * Retrieves the current {@link Direction} of the {@link Node}.
     * @return Direction + or -
     */

    public String getDirection() {
        if (sign == Direction.SEND) {
            return "+";
        }
        else
            return "-";
    }

    /**
     * Sets the {@link Node} body to {@link Message} {@code filler}
     * @param filler New {@link Message} for the Node
     */
    public void setBody(Message filler) {
        body = filler;
    }
    /**
     * Retrieves the current {@link Message} associated with this {@link Node}.
     * @return Message associated with this {@link Node}
     */
    public Message getBody() {
        return body;
    }

    /**
     * Returns the list of {@link Atom}s used in the {@link Message} at this {@link Node}.
     * @return list of {@link Atom}s used in this {@link Message}
     */
    public ArrayList<Atom> getNodeVars() {
        Atom tempAtom;
        ArrayList<Atom> nodeVars = new ArrayList<Atom>();
        Iterator<Atom> tempIter = this.getBody().getMsgVars().iterator();
        while (tempIter.hasNext()) {
            tempAtom = tempIter.next();
            if (!(nodeVars.contains(tempAtom)))
                nodeVars.add(tempAtom);
        }
        return nodeVars;
    }

    /**
     * Returns the list of unique {@link Name}s used in the {@link Message} at this {@link
     * Node}.
     * @return list of {@link Name}s used in this {@link Message}
     */
/*  public ArrayList<Name> getNodeNames() {
        Atom tempAtom;
        ArrayList<Name> nodeNames = new ArrayList<Name>();
        Iterator<Atom> tempIter = this.getBody().getMsgVars().iterator();
        while (tempIter.hasNext()) {
            tempAtom = tempIter.next();
            if ((tempAtom instanceof Name ) && (!(nodeNames.contains(tempAtom))))
                nodeNames.add((Name) tempAtom);
        }
        return nodeNames;
    }
    */
    /**
     * Returns the list of unique {@link Nonce}s used in the {@link Message} at this {@link
     * Node}.
     * @return list of {@link Nonce}s used in this {@link Message}
     */
    /*public ArrayList<Nonce> getNodeNonces() {
        Atom tempAtom;
        ArrayList<Nonce> nodeNonces = new ArrayList<Nonce>();
        Iterator<Atom> tempIter = this.getBody().getMsgVars().iterator();
        while (tempIter.hasNext()) {
            tempAtom = tempIter.next();
            if ((tempAtom instanceof Nonce ) && (!(nodeNonces.contains(tempAtom))))
                nodeNonces.add((Nonce) tempAtom);
        }
        return nodeNonces;
    }
    */
    /**
     * Returns the list of unique {@link Keys}s used in the {@link Message} at this {@link
     * Node}.
     * @return list of {@link Keys}s used in this {@link Message}
     */
    /*public ArrayList<Key> getNodeKeys() {
        Atom tempAtom;
        ArrayList<Key> nodeKeys = new ArrayList<Key>();
        Iterator<Atom> tempIter = this.getBody().
                                getMsgVars().iterator();
        while (tempIter.hasNext()) {
            tempAtom = tempIter.next();
            if ((tempAtom instanceof Key ) &&
                (!(nodeKeys.contains(tempAtom))))
                nodeKeys.add((Key) tempAtom);
        }
        return nodeKeys;
    }
    */
    /**
     * Prints the Node direction and calls the
     * {@link Message} output method for the Node body.
     */
    public void output() {
        System.out.print(this.getDirection() + " (");
        body.output();
        System.out.println(")");
    }

    /**
     * Returns the {@code Direction} and message of this
     * {@link Node} as a single {@link String}
     */
    public String toString() {
        String returnString = this.getDirection() + " ";
        returnString = returnString.concat(body.toString());
        return returnString;
    }
}
```

C.8 Message.java

```
import java.util.ArrayList;
import java.util.Iterator;
```

```
/**
 *   Abstract class representing the information sent or received on a {@link Strand}. May
 *   contain an {@link Encryption}, a {@link Concatenation}, or an atom.
 *   @author Stephanie Skaff
 *   @see Encryption, Cons, Atom
 */


public abstract class Message {
    public static ArrayList<Message> messages;
    public static int msgDepth;
    private ArrayList<Atom> msgVars;

    static {
        messages = new ArrayList<Message>();
        msgDepth = 0;
    }

    /**
     * Creates an empty {@link Message}
     */
    public Message() {
        messages.add(this);
        //msgVars = new ArrayList<Atom>();
    }

    public Message(ArrayList<Atom> vars) {
        messages.add(this);
        msgVars = vars;
    }



    public static Message realistic
        (ArrayList<Message> msgBucket, Distribution distro) {
        Message tempMsg, tempMsg2;
        Iterator<Message> bucketIter;
        boolean hasAtom = false;
        boolean hasEnc = false;
        int msgIndex;

        msgIndex = Generator.randInt.nextInt(msgBucket.size());
        tempMsg = msgBucket.get(msgIndex);
        msgBucket.remove(msgIndex);
        if (tempMsg instanceof Atom) {
            if (msgBucket.size() == 0) {
                return tempMsg;
            }

            else {
                // Bucket not empty; any encryptions?
                bucketIter = msgBucket.iterator();
            while (bucketIter.hasNext()){
                tempMsg2 = bucketIter.next();
                if (tempMsg2 instanceof Atom) {
                    hasAtom = true;
                }
                else {  // tempMsg is an Encryption
                    hasEnc = true;
                }
                if (hasAtom && hasEnc) break;
            }
            if (hasEnc && (!(hasAtom))) {
                // swap this atom for an encryption
                do {
                    msgIndex = Generator.randInt.nextInt(msgBucket.size());
                    tempMsg2 = msgBucket.get(msgIndex);
                }
                while (!(tempMsg2 instanceof Encryption));
                msgBucket.set(msgIndex, tempMsg);
                Encryption returnMsg = (Encryption) tempMsg2;
                returnMsg.setKey(Key.anyEncryptor());
                returnMsg.setContents(Message.realistic(msgBucket, distro));
                return returnMsg;
            }
            else {
                Cons returnMsg = new Cons();
                returnMsg.setFirst(tempMsg);
                returnMsg.setSecond(Message.realistic(msgBucket, distro));
                return returnMsg;
            }
        }
    }
    else {  // must be an encryption
        Encryption returnMsg = (Encryption) tempMsg;
        returnMsg.setKey(Key.anyEncryptor());
        returnMsg.setContents(Message.realistic(msgBucket, distro));
        return returnMsg;
    }
}

/**
 * Selects type of {@link Message} extension according to an unbounded naive (uniform)
 * distribution)
 */
 public static Message naive() {
     Message body;
     Message.incMsgDepth();
     double nodeRand = Math.random();
     // select Message body type
     if ((Message.getMsgDepth() > 15) ||
             (nodeRand >= .6667))  {
         body = Atom.random();
     }
     else if (nodeRand < .3334) {
        Encryption tempEnc = Encryption.random();
        if (tempEnc.getContents() instanceof Encryption) {
            while (tempEnc.equals((Encryption) tempEnc.getContents())) {
                tempEnc.setContents(Encryption.random());
            }
        }
        body = tempEnc;
    }
    else {
        Cons tempCons = Cons.random();
        if (tempCons.getSecond() instanceof Cons) {
            while (tempCons.equals((Cons) tempCons.getSecond())) {
                tempCons.setSecond(Cons.random());
            }
        }
        body = tempCons;
    }
    Message.decMsgDepth();
    return body;
}


public static Message boundedNaive() {
    Message body;
    Message.incMsgDepth();
    double nodeRand = Math.random();
    // random contents
    if ((Message.getMsgDepth() >= 15) ||
            (nodeRand > .6667)) {
        body = Atom.boundedNaive();
    }
    else if (nodeRand < .3334) {
        Encryption tempEnc = Encryption.boundedNaive();
        if (tempEnc.getContents() instanceof Encryption) {
            while (tempEnc.equals((Encryption) tempEnc.getContents())) {
                tempEnc.setContents(Encryption.boundedNaive());
            }
        }
        body = tempEnc;
    }
    else {
        Cons tempCons = Cons.boundedNaive();
        if (tempCons.getSecond() instanceof Cons) {
            while (tempCons.equals((Cons) tempCons.getSecond())) {
                tempCons.setSecond(Cons.boundedNaive());
            }
        }
        body = tempCons;
    }
    Message.decMsgDepth();
    return body;
}
```

```
public ArrayList<Atom> getMsgVars() {
    ArrayList<Atom> varList = new ArrayList<Atom>();
    Iterator<Atom> tempIter =null;
    Key tempKey;

    if (this instanceof Atom) {
        if ((this instanceof Name) || (this instanceof Nonce)) {
            varList.add((Atom) this);
        }
        else {  // Atom must be a Key
            tempKey = ((Key) this);
            varList.add(tempKey);
            tempIter = tempKey.getKeyHolders().iterator();
            while (tempIter.hasNext()) {
                varList.add(tempIter.next());
            }
        }
    }
    else if (this instanceof Cons) {
        Cons tempCons = (Cons) this;
        tempIter = tempCons.getFirst().getMsgVars().iterator();
        while (tempIter.hasNext()) {
            varList.add(tempIter.next());
        }
        tempIter = tempCons.getSecond().getMsgVars().iterator();
        while (tempIter.hasNext()) {
            varList.add(tempIter.next());
         }
    }
    else {  // must be an Encryption
        Encryption tempEnc = (Encryption) this;
        tempKey = (Key) tempEnc.getKey();
        tempIter = tempKey.getKeyHolders().iterator();
        while (tempIter.hasNext()) {
            varList.add(tempIter.next());
        }
        tempIter = tempEnc.getContents().getMsgVars().iterator();
        while (tempIter.hasNext()) {
            varList.add(tempIter.next());
         }

    }
    return varList;
}


/* ************************************************
 * Message access functions and output
 ************************************************/

/**
 * Returns the depth of the current {@code Message}
 * @return {@link int} representing current nesting depth
```

```
     */
    public static int getMsgDepth() {
        return Message.msgDepth;
    }

    /**
     * Sets the depth of the {@link Message} depth counter to the specified value
     *
     * @param newDepth new value of MsgDepth
     */
    public static void setMsgDepth(int newDepth) {
        Message.msgDepth = newDepth;
    }

    /**
     * Increments the {@code msgDepth} counter
     */
    public static void incMsgDepth() {
        Message.msgDepth++;
    }

    /**
     * Decrements the {@code msgDepth} counter
     */
    public static void decMsgDepth() {

        Message.msgDepth--;
    }

    /**
     * Gets the total number of Messages generated so far
     * @return count of {@link Message}s generated
     */
    public static int getCount() {
        return messages.size();
    }

    /**
     * Return the {@link Message} stored at position {@code index}
     * @param index position of {@link Message} to retrieve
     * @return {@link Message at position {@code index}
     */
    public static Message getMessage(int index) {
        return messages.get(index);
    }
    /*
    protected void addVar(Atom aTom) {
        msgVars.add(aTom);
    }
*/
    public abstract Message clone();

    /**
```

```
     * Calls appropriate extended class toString method for
     * this Message.
     */
    public void output() {
        if (this instanceof Atom) {
            Atom tempAtom = (Atom) this;
            System.out.print(tempAtom.toString());
        }
        else if (this instanceof Encryption) {
            Encryption tempEnc = (Encryption) this;
            tempEnc.output();
        }
        else if (this instanceof Cons) {
            Cons tempCons = (Cons) this;
            tempCons.output();
        }
    }

    /**
     * Mandatory method for converting messages to Strings;
     * must be implemented by extending class.
     */
    public abstract String toString();

}
```

C.9 Atom.java

```
/**
 *  Abstract class for Names, Nonces and Keys.
 *
 *  @author Stephanie Skaff
 */

public abstract class Atom extends Message{

    public static Atom random() {
        Atom aTom;
        double atomRand = Math.random();
        if (atomRand < .33334) {
            // message is a name - new?
            aTom = Name.random();
        }
        else if (atomRand < .66667) {
            //new or old?
            aTom = Nonce.random();
        }
        else {
            // message is a key - new?
            aTom = Key.random();
        }
        return aTom;
    }

    public static Atom boundedNaive() {
        Atom aTom;
        double atomRand = Math.random();
        if (atomRand < .33334) {
            // message is a name - new?
            aTom = Name.boundedNaive();
        }
        else if (atomRand < .66667) {
            //new or old?
            aTom = Nonce.boundedNaive();
        }
        else {
            // message is a key - new?
            //body =
            aTom = Key.boundedNaive();
        }
        return aTom;
    }

    /**
     * Overriding abstract class for converting messages to
     * Strings.
     */
    public abstract String toString();

}
```

C.10 Name.java

```
import java.util.ArrayList;
import java.util.Iterator;

/**
 * Identification of entities on a network.
 * @author Stephanie Skaff
 *
 */
public class Name extends Atom {
    private String theName;
    public static ArrayList<Name> names;

    static {
        names = new ArrayList<Name>();
    }

    /**
     * Creates an empty @link Name} and adds it the
     * {@code names} list
     */
    public Name() {
        names.add(this);
    }

    /**
     * Creates a new {@link Name} with the specified name value and adds it to the {@code
     * names} list
     * @param name  value of this {@link Name}
     */
    public Name(String name) {
        theName = name;
        names.add(this);
    }

    /**
     * Returns a random {@link Name} with uniform probability.
     * @return a random name
     */
    public static Name random() {
        Name returnName;
        double nameRand = Math.random();
        if (nameRand < .5 ) {
            //  create a new name
            returnName = new Name(Name.getNewNameString());
        }
        else {
            returnName = names.get(Generator.randInt.nextInt(names.size()));
        }
        return returnName;
    }

    /**
     * Returns a random {@link Name} with a upper-bounded
     * uniform probability.
     * @return
     */
    public static Name boundedNaive() {
        Name returnName;
        double nameRand = Math.random();
        if ((names.size() == 0) || ((names.size() < 10) && (nameRand < .5 ))) {
            //  create a new name
            returnName = new Name(Name.getNewNameString());
        }
        else {
            Name tempName = names.get(Generator.randInt.nextInt(names.size()));
            returnName = tempName.clone();
        }
        return returnName;
    }

    /**
     * Sets the ID {@code theName} for this {@link Name} to {@code name}.
```

```
 * @param name  ID for this {@link Name}
 */
public void setName(String name) {
    theName = name;
}

/**
 * Returns the ID of this {@link Name}.
 */
public String getName() {
    return theName;
}

/**
 * Returns the number of {@link Name}s created so far in this {@link Protocol}.
 * @return count of the existing {@link Name}s
 */
public static int getNameCount() {
    return names.size();
}

/**
 * Creates a String for generated {@link Name}s.
 * @return
 */
public static String getNewNameString() {
    return "gen_name" + Name.getNameCount();
}

/**
 * Tests equality of this {@link Name} with another.
 * @param tester {@link Name} to test for equality with this one
 * @return True if the two{@link Name}s are identical; false otherwise.
 */
public boolean equals(Name tester) {
    Name tempName = (Name) tester;
    return this.theName.equals(tempName.getName());
}

public static boolean checkUniqueName(Name tester, Iterator<Name> iterNames){
    boolean addIt = true;
    Name tempName;
    if (!(iterNames.hasNext())) {
        addIt = true;
    }
    else {  // check tempAtom against atoms in iterNames
        while (iterNames.hasNext()) {
            tempName = iterNames.next();
            if (tempName.equals(tester)) {
                addIt = false;
            }
        }
    }
```

```
        return addIt;
    }

    /**
     * Returns this {@link Name}.
     */
    public Name clone() {
        return this;
    }

    /**
     * Returns a String represention of a {@link Name}.
     */
    public String toString() {
        return theName;
    }
}
```

C.11 Nonce.java

```
import java.util.ArrayList;

/**
 * Represents a one-time value, created freshly for each execution of a {@link Protocol}.
 * @author Stephanie Skaff
 */
public class Nonce extends Atom{
    private int aNonce;
    private String nonceName;
    public static ArrayList<Nonce> nonces;

    static {
        nonces = new ArrayList<Nonce>();
    }

    /**
     * Empty Nonce constructor
     */
    public Nonce() {
        super();
        nonces.add(this);
        nonceName = "nonce" + nonces.size();
    }


    /**
     * Randomly returns either a new {@link Nonce} or a clone of a previously existing Nonce
     * @return a random {@link Nonce}
     */
    public static Nonce random() {
        Nonce returnNonce;
    double nonceRand = Math.random();
    if ((nonceRand < .5 ) || (nonces.size() == 0)) {
        //  create a new nonce
        returnNonce = new Nonce();
    }
    else {
        returnNonce = nonces.get(Generator.randInt.nextInt(nonces.size()));
    }
    return returnNonce;
}

public static Nonce boundedNaive() {
    Nonce returnNonce;
    double nonceRand = Math.random();
    if ((nonces.size() == 0 ) || ((nonces.size() < 10) && (nonceRand < .5 ))) {
        //  create a new nonce
        returnNonce = new Nonce();
    }
    else {
        Nonce tempNonce = nonces.get(Generator.randInt.nextInt(nonces.size()));
        returnNonce = tempNonce.clone();
    }
    return returnNonce;
}

/**
 * Sets the value of the Nonce
 * @param value Value of the Nonce
 */
public void setValue(int value) {
    aNonce = value;
}

/**
 * Returns the Nonce value
 * @return aNonce   Value of the nonce
 */
public int getValue() {
    return aNonce;
}

public void setName(String name) {
    nonceName = name;
}

/**
 * Returns the ID string for this {@link Nonce}
 * @return ID string for this {@link Nonce}
 */
public String getName() {
    return nonceName;
}
```

```
    /**
     * Equality check for {@link Nonce}s, accomplished by checking the {@link nonceName}.
     *
     * @param test {@link Nonce} against which to compare {@link Nonce} values
     * @return  Result of value equality check
     */
    public boolean equals(Nonce tester) {
        return (this.nonceName.equals(tester.nonceName));
    }


    /**
     * Returns a new Nonce with an identical value
     *
     * @return returnNonce  The new cloned Nonce
     */
    public Nonce clone() {
        return this;
    }

    /**
     * Returns a String representation of the {@link Nonce.}
     *
     * @return String of the form "nonce 73"
     */
    public String toString() {
        return nonceName;
    }
}
```

C.12 Key.java

```
import java.util.ArrayList;

/**
 * Abstract class representing cryptographic keys
 * @author Stephanie Skaff
 */
public abstract class Key extends Atom {
    public static ArrayList<Encryptor> encryptors;
    public static ArrayList<Decryptor> decryptors;
    public static ArrayList<AsymKey> asymKeys;
    public static ArrayList<SymKey> symKeys;
    public static ArrayList<Key> allKeys;

    static {
        encryptors = new ArrayList<Encryptor>();
        decryptors = new ArrayList<Decryptor>();
        asymKeys = new ArrayList<AsymKey>();
        symKeys = new ArrayList<SymKey>();
        allKeys = new ArrayList<Key>();
```

```
}

//constructors
public Key() {
    allKeys.add(this);
}


public ArrayList<Atom> getKeyHolders() {
    ArrayList<Atom> keyholders = new ArrayList<Atom>();
    if (this instanceof AsymKey) {
         AsymKey tempKey = (AsymKey) this;
         keyholders.add(tempKey.getKeyholder());
     }
     else if (this instanceof SymKey) {
         SymKey tempKey = (SymKey) this;
         keyholders.add(tempKey.getKeyholder1());
         keyholders.add(tempKey.getKeyholder2());
     }
    return keyholders;
}

/**
 * Selects a random {@link Key} to transmit
 * @return selected random {@link Key}
 */
public static Key random() {
    double encKey = Math.random();
    if (encKey < .33334) {
        //encrypt with a PublicKey
        return PublicKey.random();
    }
    else if (encKey < .66667) {
        // encrypt with a sessionKey
        return SessionKey.random();
    }
    else {
        //return with a long term pair key
        return LTPairKey.random();
    }
}

/**
 * Selects a random {@link Key} to transmit. This is
 * identical in function to Key.random(), as the bounds
 * are at the more specific key level.
 * @return selected random {@link Key}
 */
public static Key boundedNaive() {
    double encKey = Math.random();
    if (encKey < .33334) {
        //encrypt with a PublicKey
        return PublicKey.random();
    }
    else if (encKey < .66667) {
        // encrypt with a sessionKey
        return SessionKey.random();
    }
    else {
        //return with a long term pair key
        return LTPairKey.random();
    }
}

/**
 * Returns the current Key, as keys should not be cloned. Implements Message.clone().
 *
 * @return The current Key object
 */
public Key clone() {
    return this;
}

/**
 * Returns count of all existing keys
 * @return size Number of keys in existence
 */
public static int allKeyCount() {
    return allKeys.size();
}

/**
 * Returns the encryption key stored at {@code index}.
 * @param index Location in encKeys list
 * @return  Key stored at specified index
 */
public static Key getAllKey(int index) {
    return allKeys.get(index);
}

/**
 * Returns count of existing encryption keys
 * @return size Number of encryption keys in existence
 */

public static int encKeyCount() {
    return encryptors.size();
}


/**
 * Returns the encryption key stored at {@code index}.
 * @param index Location in encKeys list
 * @return  Key stored at specified index
 */
public static Encryptor getEncKey(int index) {
        return encryptors.get(index);
    }

    /**
     * Returns count of existing asymmetric keys
     * @return size Number of decryption keys in existence
     */
    public static int asymKeyCount() {
        return asymKeys.size();
    }

    /**
     * Returns the asymmetric key stored at {@code index}.
     * @param index Location in decKeys list
     * @return  Key stored at specified index
     */

    public static AsymKey getAsymKey(int index) {
        return asymKeys.get(index);
    }

    /**
     * Returns count of existing asymmetric keys
     * @return size Number of decryption keys in existence
     */
    public static int SymKeyCount() {
        return symKeys.size();
    }

    /**
     * Returns the asymmetric key stored at {@code index}.
     * @param index Location in decKeys list
     * @return  Key stored at specified index
     */

    public static SymKey getSymKey(int index) {
        return symKeys.get(index);
    }
}
```

C.13 Encryptor.java

```
/**
 * This interface is used to designate keys which can encrypt {@link Message}s.  It has no
 * separate functions at this time, but exists only for {@link instanceof} checks.
 * @author Stephanie Skaff
 */

public interface Encryptor {}
```

C.14 Decryptor.java

```
/**
 * This interface is used to designate keys which can decrypt {@link Message}s.  It has no
 * separate functions at this time, but exists only for {@link instanceof} checks.
 * @author Stephanie Skaff
 */

public interface Decryptor {}
```

C.15 SymKey.java

```
/**
 *  Symmetric Key interface, implemented by {@link LTPairKey} and {@link SessionKey}
 *  @author Stephanie Skaff
 */

public interface SymKey {

    public Name getKeyholder1();

    public Name getKeyholder2();

    public boolean sharesThisKey(Name aName);
}
```

C.16 LTPairKey.java

```
import java.util.ArrayList;

/**
 * Represents a pre-existing shared (secret) key between two network entities (as defined
 * by their {@link Name}s).These participants may or may not be protocol participants/
 * {@link Strand}s.
 *
 * Implements: Encryptor, Decryptor, SymKey
 * @author Stephanie Skaff
 */
public class LTPairKey extends Key
                       implements Encryptor, Decryptor, SymKey {
    private Name keyholder1;
    private Name keyholder2;
    public static ArrayList<LTPairKey> ltPairKeys;

    static {
        ltPairKeys = new ArrayList<LTPairKey>();
    }


    /**
 * Creates a long-term shared key between {@code name1} and {@code name.2}
 * @param name1 Name of {@code keyholder1}
 * @param name2 Name of {@code keyholder2}
 */
public LTPairKey(Name name1, Name name2) {
    super();
    keyholder1 = name1;
    keyholder2 = name2;
    ltPairKeys.add(this);
    encryptors.add(this);
    decryptors.add(this);
    symKeys.add(this);
}

/**
 * Returns a random (unbounded uniform) long-term shared key
 * @return randomly selected key
 */
public static LTPairKey random() {
    LTPairKey returnLTPKey;
    Name key1, key2;
    double ltpRand = Math.random();
    if ((ltPairKeys.size() == 0) ||(ltpRand <.5)) {
        /*  making a new session key; is either
            participant new?  */
        ltpRand = Math.random();
        if ((Name.names.size() == 0) ||(ltpRand <.5)) {
            key1 = new Name(Name.getNewNameString());
        }
        else {
            key1 = Name.names.get(Generator.randInt.nextInt(Name.names.size()));
        }
        ltpRand = Math.random();
        if ((Name.names.size() == 0) ||(ltpRand <.5)) {
            key2 = new Name(Name.getNewNameString());
        }
        else {
            key2 = Name.names.get(Generator.randInt.nextInt(Name.names.size()));
        }
        returnLTPKey = new LTPairKey(key1, key2);
    }
    else {
        //pick a random existing session key
        returnLTPKey = ltPairKeys.get(Generator.randInt.nextInt(ltPairKeys.size()));
    }
    return returnLTPKey;
}

/**
 * Returns a random (bounded uniform) long-term shared key
 * @return randomly selected key
 */
public static LTPairKey boundedNaive() {
        LTPairKey returnLTPKey;
        Name key1, key2;
        double ltpRand = Math.random();
        if ((ltPairKeys.size() == 0) || ((ltPairKeys.size() < 10) && (ltpRand <.5))) {
            /*  making a new session key; is either participant new?  */
            ltpRand = Math.random();
            if ((Name.names.size() == 0) || ((Name.names.size() < 10) && (ltpRand <.5))) {
                key1 = new Name(Name.getNewNameString());
            }
            else {
                key1 = Name.names.get(Generator.randInt.nextInt(Name.names.size()));
            }
            ltpRand = Math.random();
            if ((Name.names.size() == 0) || ((Name.names.size() < 10) && (ltpRand <.5))) {
                key2 = new Name(Name.getNewNameString());
            }
            else {
                key2 = Name.names.get(Generator.randInt.nextInt(Name.names.size()));
            }
            returnLTPKey = new LTPairKey(key1, key2);
        }
        else {
            //pick a random existing session key
            returnLTPKey = ltPairKeys.get(Generator.randInt.nextInt(ltPairKeys.size()));
        }
        return returnLTPKey;
    }

    /**
     * Returns the {@link Name} of the first keyholder
     */
    public Name getKeyholder1() {
        return keyholder1;
    }

    /**
     * Returns the {@link Name} of the second keyholder
     */
    public Name getKeyholder2() {
        return keyholder2;

    }

    /**
     * Determines whether {@code aName} is one of the keyholders for this key.
     */
    public boolean sharesThisKey(Name aName) {
        if ((keyholder1.getName() == aName.getName()) ||
                (keyholder2.getName() == aName.getName()))
            return true;
        else
            return false;
            }
    /**
     * Determines whether this key is equal to {@code ltpk}; this is determined by comparing
     * the identities of the keyholders.
     * @param ltpk
     * @return
     */
    public boolean equals(LTPairKey ltpk) {
        if ((keyholder1.getName() == ltpk.getKeyholder1().getName()) &&
             (keyholder2.getName() == ltpk.getKeyholder2().getName()))
            return true;
        else
            return false;
            }

    public Key clone() {
        return this;
    }

    /**
     * Returns a {@code String} representation of this key.
     */
    public String toString() {
        return "(ltk " + keyholder1.getName() + " " + keyholder2.getName() + ")";
    }
}
```

C.17 SessionKey.java

```
import java.util.ArrayList;

/**
 * Represents a key created for temporary use between two {@link Name}s.
 *
 * @author Stephanie Skaff
 * @see {@link Key}, {@link Encryptor}, {@link Decryptor},
 * {@link SymKey}
 */

public class SessionKey extends Key
                implements Encryptor, Decryptor, SymKey {
    private Name keyholder1;
    private Name keyholder2;
    public static ArrayList<SessionKey> sessionKeys;

    static {
        sessionKeys = new ArrayList<SessionKey>();
    }


    /**
 * Creates a SessionKey not bound to any {@link Name}s
 */

public SessionKey() {
    super();
    sessionKeys.add(this);
    encryptors.add(this);
    decryptors.add(this);
    symKeys.add(this);
}

/**
 * Creates a SessionKey between {@code name1} and {@code name2}
 *
 * @param name1 first {@link Name} sharing this key
 * @param name2 second {@link Name} sharing this key
 */

public SessionKey(Name name1, Name name2) {
    super();
    keyholder1 = name1;
    keyholder2 = name2;
    sessionKeys.add(this);
    encryptors.add(this);
    decryptors.add(this);
    symKeys.add(this);
}

/**
 * Returns a random SessionKey.  This SessionKey may be either already existing or
 * freshly created with uniform probability.
 *
 * @return a random SessionKey
 */
public static SessionKey random() {
    SessionKey returnSessionKey;
    Name key1, key2;
    double sessionRand = Math.random();
    if ((sessionKeys.size() == 0) ||(sessionRand <.5)) {
        //  making a new session key; either player new?
        sessionRand = Math.random();
        if ((Name.names.size() == 0) || (sessionRand <.5)) {
            key1 = new Name(Name.getNewNameString());
        }
        else {
            key1 = Name.names.get(Generator.randInt.nextInt(Name.names.size()));
        }
        sessionRand = Math.random();
        if ((Name.names.size() <= 1) || (sessionRand <.5)) {
            key2 = new Name(Name.getNewNameString());
        }
        else {
            key2 = Name.names.get(Generator.randInt.nextInt(Name.names.size()));
        }
        returnSessionKey = new SessionKey(key1, key2);
    }
    else {
        //pick a random existing session key
        returnSessionKey = sessionKeys.get(Generator.randInt.nextInt(sessionKeys.size()));
    }
    return returnSessionKey;
}

/**
 * Returns a random SessionKey with uniform probability
 * within upper bounds.
 * @return a random {@link SessionKey}
 */
public static SessionKey boundedNaive() {
    SessionKey returnSessionKey;
    Name key1, key2;
    double sessionRand = Math.random();
    if ((sessionKeys.size() == 0) || (sessionKeys.size() < 10 && (sessionRand <.5))) {
        //  making a new session key; either player new?
        sessionRand = Math.random();
        if ((Name.names.size() == 0) || ((Name.names.size() < 10) && (sessionRand <.5))) {
            key1 = new Name(Name.getNewNameString());
        }
        else {
            key1 = Name.names.get(Generator.randInt.nextInt(Name.names.size()));
        }
        sessionRand = Math.random();
        if ((Name.names.size() <= 1) || ((Name.names.size() < 10) && (sessionRand <.5))) {
            key2 = new Name(Name.getNewNameString());
        }
        else {
            key2 = Name.names.get(Generator.randInt.nextInt(Name.names.size()));
        }
        returnSessionKey = new SessionKey(key1, key2);
    }
    else {
        //pick a random existing session key
        returnSessionKey = sessionKeys.get(Generator.randInt.nextInt(sessionKeys.size()));
    }
    return returnSessionKey;
}

/**
 * Returns the first {@link Name} associated with this SessionKey.
 */

public Name getKeyholder1() {
    return keyholder1;
}

/**
```

```
     * Returns the second {@link Name} associated with this SessionKey.
     */

    public Name getKeyholder2() {
        return keyholder2;
    }

    /**
     * Checks whether the supplied {@link Name} is associated with this SessionKey.
     *
     *  @param aName {@link Name} to check for association with this key
     */
    public boolean sharesThisKey(Name aName) {
        if ((keyholder1.getName() == aName.getName()) ||
            (keyholder2.getName() == aName.getName()))
            return true;
        else
            return false;
            }

    /**
     * Returns a link to this SessionKey.
     */
    public Key clone() {
        return this;
    }

    /**
     * Outputs the SessionKey as a {@code String} containing
     * its associated {@link Name}s.
     */
    public String toString() {
        return "(sessionkey " + keyholder1.getName() + " " + keyholder2.getName() + ")";
    }
}
```

C.18 AsymKey.java

```
/**
 *  Asymmetric Key interface
 *  @author Stephanie Skaff
 */

public interface AsymKey {

    public Name getKeyholder();

    public void setKeyholder(Name aName);

    public boolean ownsThisKey(Name aName);
}
```

C.19 PublicKey.java

```
import java.util.ArrayList;

/**
 * Represents the public half of an {@link AsymKey} pair.
 *
 * @author stephanie
 *
 */
public class PublicKey extends Key implements Encryptor, AsymKey {

    private Name keyholder;
    public static ArrayList<PublicKey> pubKeys;

    static {
        pubKeys = new ArrayList<PublicKey>();
    }

    /**
     * Creates a {@link PublicKey} not associated with a {@link Name}
     */

    public PublicKey() {
        super();
        pubKeys.add(this);
        encryptors.add(this);
        asymKeys.add(this);
    }

    /**
     * Creates a {@link PublicKey} associated with {@code aName}
     * @param aName {@link Name} to be associated with this key
     */
    public PublicKey(Name aName) {
        super();
        keyholder = aName;
        pubKeys.add(this);
        encryptors.add(this);
    }

    /**
     * Returns a random {@link PublicKey} with uniform probability
     * @return a random {@link PublicKey}
     */
    public static PublicKey random() {
        PublicKey returnPubKey;
        double pubRand = Math.random();
    if ((pubKeys.size() == 0) || (pubRand < .5 )) {
        //  create a new Name and associated keys
        Name newName = new Name(Name.getNewNameString());
        returnPubKey = pubKeyOf(newName);
    }
    else {
        returnPubKey = pubKeys.get(Generator.randInt.nextInt(pubKeys.size()));
    }
    return returnPubKey;
}


/**
 * Returns a random {@link PublicKey} according to uniform probability with upper bounds
 * @return a random {@link PublicKey}
 */
public static PublicKey boundedNaive() {
    PublicKey returnPubKey;
    double pubRand = Math.random();

    if ((pubKeys.size() == 0) || (pubKeys.size() < 10 && (Name.names.size() < 10)
                                                    && (pubRand < .5 ))) {
        //  create a new Name and associated key
        Name newName = new Name(Name.getNewNameString());
        returnPubKey = pubKeyOf(newName);
    }
    else {
        returnPubKey = pubKeys.get(Generator.randInt.nextInt(pubKeys.size()));
    }
    return returnPubKey;
}

/**
 * Assign this {@link PublicKey} to the provided {@link Name}.
 * @param aName {@link Name} of the owner of this {@link PublicKey}
 */
public void setKeyholder(Name aName) {
    keyholder = aName;
}

/**
 * Returns the {@link Name} associated with the keyholder of this {@link PublicKey}.
 * @return the keyholder's {@link Name}
 */
public Name getKeyholder() {
    return keyholder;
}

/**
 * Checks whether {@code aName} is the owner of this {@link PublicKey}
 * @return true if {@code aName} owns this key, otherwise false
 */
```

```
    public boolean ownsThisKey(Name aName) {
        if (keyholder.getName() == aName.getName())
            return true;
        else
            return false;
    }

    /**
     * Returns the {@link PublicKey} object associated with the supplied {@link Name}.
     * @param aName {@link Name} whose {@link PublicKey} is needed
     * @return the {@link PublicKey} associated with {@code aName}
     */
    public static PublicKey pubKeyOf(Name aName) {
        int keyAt = -1;
        PublicKey returnKey;
        for (int i = 0; i < pubKeys.size(); i++) {
            if (pubKeys.get(i).keyholder.equals(aName)) {
                keyAt = i;
            }
            if (keyAt != -1) break;
        }
        if (keyAt != -1) {
            returnKey = pubKeys.get(keyAt);
        }
        else
            returnKey = new PublicKey(aName);
        return returnKey;
    }

    /**
     * Returns this {@link PublicKey}
     */
    public Key clone() {
        return this;
    }

    /**
     * Returns a String representation of this {@link
     * PublicKey}.
     */
    public String toString() {
        return "(pubkey " + keyholder.getName() + ")";
    }
}
```

C.20 PrivateKey.java

```
import java.util.ArrayList;

/**
 * Represents the private half of an {@link AsymKey} pair.
 * @author Stephanie Skaff
 */
public class PrivateKey extends Key implements Decryptor, AsymKey {

    private Name keyholder;
    public static ArrayList<PrivateKey> privKeys;

    static {
        privKeys = new ArrayList<PrivateKey>();
    }

    /**
     * Creates a {@link PrivateKey} not associated with a {@link Name}
     */
    public PrivateKey() {
        super();
        privKeys.add(this);
        decryptors.add(this);
        asymKeys.add(this);
    }

    /**
     * Creates a {@link PrivateKey} associated with {@code aName}
     * @param aName {@link Name} to be associated with this key
     */
    public PrivateKey(Name aName) {
        super();
        keyholder = aName;
        privKeys.add(this);
        decryptors.add(this);

    }

    /**
     * Returns a random {@link PrivateKey} with uniform probability
     * @return a random {@link PrivateKey}
     */
    public static PrivateKey random() {
        PrivateKey returnPrivKey;
        double privRand = Math.random();
        if (privRand < .5 ) {
            //  create a new Name and associated keys
            Name newName = new Name(Name.getNewNameString());
            returnPrivKey = privKeyOf(newName);
        }
        else {
            returnPrivKey = privKeys.get(Generator.randInt.nextInt(privKeys.size()));
        }
        return returnPrivKey;
    }

    /**
     * Returns a random {@link PrivateKey} according to uniform probability with upper bounds
 * @return a random {@link PrivateKey}
 */
public static PrivateKey boundedNaive() {
    PrivateKey returnPrivKey;
    double privRand = Math.random();
    if (privRand < .50 ) {
        //  create a new Name and associated keys
        Name newName = new Name(Name.getNewNameString());
        returnPrivKey = privKeyOf(newName);
    }
    else {
        returnPrivKey = privKeys.get(Generator.randInt.nextInt(privKeys.size()));
    }
    return returnPrivKey;
}

/**
 * Assign this {@link PrivateKey} to the provided {@link Name}.
 * @param aName {@link Name} of the owner of this {@link PrivateKey}
 */
public void setKeyholder(Name aName) {
    keyholder = aName;
}

/**
 * Returns the {@link Name} associated with the keyholder of this {@link PrivateKey}.
 * @return the keyholder's {@link Name}
 */
public Name getKeyholder() {
    return keyholder;
}

/**
 * Checks whether {@code aName} is the owner of this {@link PrivateKey}
 * @return true if {@code aName} owns this key, otherwise false
 */
public boolean ownsThisKey(Name aName) {
    if (keyholder.getName() == aName.getName())
        return true;
    else
        return false;
}

/**
 * Returns the {@link PrivateKey} object associated with the supplied {@link Name}.
 * @param aName {@link Name} whose {@link PrivateKey} is needed
 * @return the {@link PrivateKey} associated with {@code aName}
 */
public static PrivateKey privKeyOf(Name aName) {
    PrivateKey returnPrivKey;
    int keyAt = -1;
    for (int i = 0; i < privKeys.size(); i++) {
        if (privKeys.get(i).keyholder.equals(aName)) {
                keyAt = i;
            }
            if (keyAt != -1) break;
        }
        if (keyAt != -1) {
            returnPrivKey = privKeys.get(keyAt);
        }
        else
            returnPrivKey = new PrivateKey(aName);
        return returnPrivKey;
    }

    /**
     * Returns a String representation of this {@link PrivateKey}.
     */
    public String toString() {
        return "(priv" + keyholder.getName() + ")";
    }
}
```

C.21 Cons.java

```
/**
 * Creates a concatenation of two {@link Message}s.
 * @author Stephanie Skaff
 *
 */
public class Cons extends Message {
    private Message message1;
    private Message message2;

    /**
     * Creates an empty {@link Cons}
     */
    public Cons() {
        super();
    }

    /**
     * Creates a {@link Cons} containing {@code stuff1} and {@code stuff2}.
     * @param stuff1 intended contents of the first part of the {@link Cons}
     * @param stuff2 intended contents of the second part of the {@link Cons}
     */
    public Cons(Message stuff1,Message stuff2) {
        super();
        this.message1 = stuff1;
        this.message2 = stuff2;
    }

    /**
     * Creates a new {@link Cons} {@link Message} with contents filled according to a
 * uniform distribution.
 * @return a newly filled {@link Cons}
 */
public static Cons random() {
    Cons returnMessage = new Cons();
    Message tempMsg = Message.naive();
    returnMessage.setFirst(tempMsg);
    tempMsg = Message.naive();
    returnMessage.setSecond(tempMsg);
    return returnMessage;
}

/**
 * Returns a new {@link Cons} filled according to an upper-bounded uniform distribution.
 * @return a newly filled {@link Cons}
 */
public static Cons boundedNaive() {
    Message tempMsg;
    Cons returnMessage = new Cons();
    Double secondhalf = Math.random();
    if ((secondhalf < .5 ) || (messages.size() > 15))
        tempMsg = Atom.boundedNaive();
    else
        tempMsg = Message.boundedNaive();
    returnMessage.setFirst(tempMsg);
    tempMsg = Message.boundedNaive();
    returnMessage.setSecond(tempMsg);

    return returnMessage;
}

/**
 * Checks equality of the contents of this {@link Cons} against {@code tester}.
 * @param tester {@link Cons} against which to check this for equality
 * @return true if the contents of the {@link Cons} objects match; otherwise false
 */
public boolean equals(Cons tester) {
    if ((this.getFirst().equals(tester.getFirst())) &&
        (this.getSecond().equals(tester.getSecond()))) {
        return true;
    }
    else return false;
}

/**
 * Creates a new {@link Cons} with identical contents
 */
public Cons clone() {
    Cons returnCons = new Cons();
    returnCons.setFirst(this.message1);
    returnCons.setSecond(this.message2);
    //returnCons.setFirst(message1.clone());
    //returnCons.setSecond(message2.clone());
```

```
        return returnCons;
    }

    /**
     * Sets first {@link Message} in this {@link Cons}
     * @param incoming desired contents for first part of this {@link Cons}
     */
    public void setFirst(Message incoming) {
        this.message1 = incoming;
    }

    /**
     * Returns first message object in this {@link Cons}.
     */
    public Message getFirst() {
        return message1;
    }

    /**
     * Sets second {@link Message} in this {@link Cons}
     * @param incoming desired contents for second part of this {@link Cons}
     */
    public void setSecond(Message incoming) {
        this.message2 = incoming;
    }

    /**
     * Returns first message object in this {@link Cons}.
     */
    public Message getSecond() {
        return message2;
    }

    /**
     * Returns String representation of concatenated message
     */
    public String toString() {
        String returnString = "(cons" + message1.toString() + "," + message2.toString() + ")";
        return returnString;
    }


    public void output() {
        System.out.print("(cons " + message1.toString() + ",");
        message2.output();
        System.out.print(")");

    }
}
```

C.22 Encryption.java

```java
/**
 * A {@link Message} encrypted by a {@link Key}.
 *
 * @author Stephanie Skaff
 * @param key Key which encrypts the accompanying Message
 * @param contents Protected contents ({@link Message}) of the encryption
 */


public class Encryption extends Message{
    private Key encKey;
    private Message contents;


    /* ***************************************************
     * Constructors
     ****************************************************/

    /**
     * Creates an empty and unkeyed {@link Encryption}
     */
    public Encryption() {
    }

    /**
     * Creates an {@link Encryption} containing a {@link Message} {@code contents} and
     * locked with {@link Key} {@code lockIt}
     * @param stuff contents of this {@link Encryption}
     * @param lockIt {@link Key} used to lock this {@link Encryption}
     */
    public Encryption(Message stuff, Key lockIt) {
        this.contents = stuff;
        this.encKey = lockIt;
    }

    /**
     * Creates a new {@link Encryption} according to an unbounded naive distribution
     * @return Encrypted {@link Message}
     */
    public static Encryption random() {
        Encryption returnEncrypt = new Encryption();
        double encRand = Math.random();
        // random contents
        if (encRand < .5) {
            returnEncrypt.setContents(Message.naive());
        }
        else {
            Message tempMsg = messages.get(Generator.randInt.nextInt(messages.size()));
            returnEncrypt.setContents(tempMsg);
        }
        encRand = Math.random();
        if (encRand < .33334) {
            //encrypt with a PublicKey
            returnEncrypt.setKey(PublicKey.random());
        }
        else if (encRand < .66667) {
            // encrypt with a sessionKey
            returnEncrypt.setKey(SessionKey.random());
        }
        else {
            //encrypt with a long term pair key
            returnEncrypt.setKey(LTPairKey.random());
        }

        return returnEncrypt;
    }

    /**
     * Creates a new {@link Encryption} according to a bounded naive distribution
     * @return Encrypted {@link Message}
     */
    public static Encryption boundedNaive() {
        Encryption returnEncrypt = new Encryption();
        double encRand = Math.random();
        // random contents
        if ((encRand < .8) || (messages.size() > 10)) {
            if (encRand < .10)
                returnEncrypt.setContents(Message.boundedNaive());
            else
                returnEncrypt.setContents(Atom.boundedNaive());
        }
        else {
            Message tempMsg = messages.get(Generator.randInt.nextInt(messages.size()));
            returnEncrypt.setContents(tempMsg);
        }
        encRand = Math.random();
        if (encRand < .33334) {
            //encrypt with a PublicKey
            returnEncrypt.setKey(PublicKey.boundedNaive());
        }
        else if (encRand < .66667) {
            // encrypt with a sessionKey
            returnEncrypt.setKey(SessionKey.boundedNaive());
        }
        else {
            //encrypt with a long term pair key
            returnEncrypt.setKey(LTPairKey.boundedNaive());
        }

        return returnEncrypt;
    }

    /**
     * Tests whether two {@link Encryption} objects are equal.
     * @param tester    Encryption for comparison
     * @return  result of equality check
     */
    public boolean equals(Encryption tester) {
        if ((this.getKey() == tester.getKey()) &&
            (this.getContents() == tester.getContents()))
            return true;
        else
            return false;
    }

    public Encryption clone() {
        Encryption returnEncrypt = new Encryption();
        returnEncrypt.setKey(this.getKey());
        returnEncrypt.setContents(this.getContents());
        return returnEncrypt;
    }
    public void setKey(Key lockIt) {
        this.encKey = lockIt;
    }

    /**
     * Returns the encrypting Key object
     */
    public Key getKey() {
        return this.encKey;
    }

    public void setContents(Message stuff) {
        this.contents = stuff;
    }

    /**
     * Returns the actual Message object stored in the encryption
     */
    public Message getContents() {
        return contents;
    }

    /**
     * Returns tagged String representation of the encryption
     */
    public String toString() {
        String returnString = "(enc" + contents.toString() + "," + encKey.toString() + ")";
        return returnString;
    }

    public void output() {
        System.out.print("(enc ");
        contents.output();
        System.out.print(encKey.toString() + ")");
    }
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] H. Wu, “The Misuse of RC4 in Microsoft Word and Excel,” *Cryptology ePrint Archive*, 2005.

[2] “CVE-2008-0166: Random number generator flaw in Debian-based OpenSSL 0.9.8,” 2008. The MITRE Corporation.

[3] R. M. Needham and M. D. Schroeder, “Using encryption for authentication in large networks of computers,” *Commununications of the ACM*, vol. 21, no. 12, pp. 993–999, 1978.

[4] G. Lowe, “An attack on the Needham-Schroeder public-key authentication protocol,” *Information Processing Letters*, vol. 56, pp. 131–133, November 1995.

[5] W. Mao and C. Boyd, “Towards formal analysis of security protocols,” in *CSFW ’93: Proceedings of the 6th IEEE Computer Security Foundations Workshop*, pp. 147–158, IEEE Computer Society, 1993.

[6] D. Otway and O. Rees, “Efficient and timely mutual authentication,” *ACM SIGOPS Operating Systems Review*, vol. 21, no. 1, pp. 8–10, 1987.

[7] “CVE-2005-2969: SSL/TLS server implementation flaw in OpenSSL 0.9.7 and 0.9.8,” 2005. The MITRE Corporation.

[8] J. A. Clark and J. L. Jacob, “Protocols are programs too: the metaheuristic search for security protocols,” *Information and Software Technology*, vol. 43, pp. 891–904, December 2001.

[9] X. D. Song, "An automatic approach to building secure systems." Ph.D. dissertation, University of California at Berkeley, Fall 2002.

[10] H. Zhou and S. N. Foley, "Fast automatic synthesis of security protocols using backward search," in *FMSE '03: Proceedings of the 2003 ACM Workshop on Formal Methods in Security Engineering*, (New York, New York, USA), pp. 1–10, ACM, 2003.

[11] H. Xue, H. Zhang, and S. Qing, "A schema of automated design security protocols," in *CISW '07: Proceedings of the 2007 Conference on Computational Intelligence and Security Workshops*, (Los Alamitos, CA, USA), pp. 733–736, IEEE Computer Society, 2007.

[12] J. A. Clark and J. L. Jacob, "A Survey of Authentication Protocol Literature, Version 1.0." Unpublished manuscript, November 1997.

[13] S. F. Doghmi, J. D. Guttman, and F. J. Thayer, *Searching for Shapes in Cryptographic Protocols*, vol. 4424 of *Lecture Notes in Computer Science*, pp. 523–537. Heidelberg, Germany: Springer Berlin, 2007.

[14] J. A. Clark and J. L. Jacob, "Searching for a solution: Engineering tradeoffs and the evolution of provably secure protocols," in *Proceedings of the 2000 IEEE Symposium on Research in Security and Privacy*, pp. 82–95, IEEE Computer Society, 2000.

[15] M. Burrows, M. Abadi, and R. Needham, "A logic of authentication," *ACM Transactions on Computer Systems*, vol. 8, no. 1, pp. 18–36, 1990.

[16] C. R. Reeves, *Modern Heuristic Techniques for Combinatorial Problems*. New York, New York, USA: Halsted Press, 1993.

[17] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Boston, Massachusetts, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[18] C. Hao, J. A. Clark, and J. L. Jacob, "Automated design of security protocols," *Computational Intelligence*, vol. 20, pp. 503–516, August 2004.

[19] A. Perrig and D. Song, "A first step to the automatic generation of security protocols," in *NDSS '00: Proceedings of the 2000 Symposium on Network and Distributed Systems Security*, (Reston, Virgina, USA), The Internet Society, February 2000.

[20] F. J. T. Fábrega, J. C. Herzog, and J. D. Guttman, "Strand spaces: Proving security protocols correct," *Journal of Computer Security*, vol. 7, no. 2-3, pp. 191–230, 1999.

[21] D. Song, S. Berezin, and A. Perrig, "Athena: a novel approach to efficient automatic security protocol analysis," *Journal of Computer Security*, vol. 9, no. 1/2, pp. 47–74, 2001.

[22] L. Buttyán, S. Staamann, and U. Wilhelm, "A simple logic for authentication protocol design," in *CSFW '98: Proceedings of the 11th IEEE Computer Security Foundations Workshop*, pp. 153–162, IEEE Computer Society, 1998.

[23] J. D. Guttman, "Security protocol design via authentication tests," in *CSFW '02: Proceedings of the 15th Computer Security Foundations Workshop*, pp. 92–103, IEEE Computer Society, 2002.

[24] N. Durgin, J. Mitchell, and D. Pavlovic, "A compositional logic for protocol correctness," in *CSFW '01: Proceedings of the 14th IEEE Computer Security Foundations Workshop*, (Washington, DC, USA), pp. 241–255, IEEE Computer Society, 2001.

[25] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, parts I and II," *Information and Computation*, vol. 100, no. 1, pp. 1–40, 1992.

[26] J. K. Millen and G. Denker, "CAPSL and MuCAPSL," *Journal of Telecommunications and Information Technology*, vol. 4, pp. 16–27, 2002.

[27] Y. Chevalier, L. Compagna, J. Cuellar, P. H. Drielsma, J. Mantovani, S. Mödersheim, and L. Vigneron, "A high level protocol specification language for industrial security-sensitive protocols," in *SAPS '04: Proceedings of the 2004 Austrian Computer Society Conference on Specification and Automated Processing of Security Requirements*, Austrian Computer Society, 2004.

[28] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. IT-29, pp. 198–208, March 1983.

[29] L. Gong, R. Needham, and R. Yahalom, “Reasoning about belief in cryptographic protocols,” *Proceedings of the 1990 IEEE Computer Society Symposium on Research in Security and Privacy*, pp. 234–248, May 1990.

[30] B. Blanchet, “An efficient cryptographic protocol verifier based on Prolog rules,” in *CSFW ’01: Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pp. 82–96, IEEE Computer Society, 2001.

[31] G. Lowe, “Casper: a compiler for the analysis of security protocols,” in *CSFW ’97: Proceedings of the 10th IEEE Computer Security Foundations Workshop*, pp. 18–30, IEEE Computer Society, 1997.

[32] D. Santhoshi and D. Shreyas, “Automated BAN analysis of authentication protocols,” graduate term paper, University of California, Irvine, Department of Information and Computer Science, 2001.

[33] M. Abadi and A. D. Gordon, “A calculus for cryptographic protocols: the spi calculus,” in *CCS ’97: Proceedings of the 4th ACM Conference on Computer and Communications Security*, (New York, New York, USA), pp. 36–47, ACM, 1997.

[34] A. D. Gordon and A. Jeffrey, “Authenticity by typing for security protocols,” in *CSFW ’01: Proceedings of the 14th IEEE Computer Security Foundations Workshop*, (Los Alamitos, CA, USA), pp. 145–159, IEEE Computer Society, 2001.

[35] C. Haack and A. Jeffrey, “Pattern-matching spi-calculus,” *Information and Computation*, vol. 204, no. 8, pp. 1195–1263, 2006.

Referenced Authors

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Edward Zieglar
   National Security Agency
   Ft. Meade, Maryland

4. Joshua Guttman
   MITRE
   Bedford, Massachussetts

5. Sylvan Pinsky
   SRI International
   Arlington, Virginia

6. Cathy Meadows
   U.S. Naval Research Laboratory
   Washington, District of Columbia

7. Jason Rogers
   U.S. Naval Research Laboratory
   Washington, District of Columbia